VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Analytické programování v jazyce C#

Analytic Programming in C# Language

2014

Bc. Lumír Kojecký

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Lumír Kojecký**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Analytické programování v jazyce C#
Analytic Programming in C# Language

Zásady pro vypracování:

Cílem práce je vytvoření analytického programování v jazyce C#, který je ekvivalentem genetického programování. V rámci laboratoře oboru navrhněte praktickou ukázku využívající produkt (např. optimalizovaný matematický model nebo naučenou neuronovou síť). Charakter práce spadá do oblasti programování v oblasti pokročilých evolučních technik.

1. Seznámení se s problematikou analytického programování.
2. Vytvoření algoritmu analytického programování.
3. Provedení testování na vybraných problémech.
4. Výsledky diskutujte v závěru.

Seznam doporučené odborné literatury:

[1] Koza J.R. 1998, Genetic Programming, MIT Press, ISBN 0-262-11189-6, 1998
[2] Koza J.R.,Bennet F.H., Andre D., Keane M. 1999, Genetic Programming III, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999
[3] Lampinen Jouni, Zelinka, Ivan, New Ideas in Optimization & Mechanical Engineering Design Optimization by Differential Evolution. Volume 1. London: McGraw-Hill, 1999. 20 p. ISBN 007-709506-5
[4] Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, STU Bratislava, ISBN 85-246-2000, 2000
[5] Zelinka I.: Analytic Programming by Means of Soma Algorithm. ICICIS'02, First International Conference on Intelligent Computing and Information Systems, Egypt, Cairo, 2002
[6] Zelinka Ivan, Evoluční výpočetní techniky - principy a alikace, BEN, Praha, 2008

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. Ing. Ivan Zelinka, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014 .....

V souvislosti s vypracováním této práce bych rád poděkoval prof. Ing. Ivanu Zelinkovi, Ph.D. za odborné vedení práce, cenné rady v oblasti analytického programování a evolučních algoritmů a v neposlední řadě za připomínky a návrhy, které pomáhaly dotvářet tuto práci.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Poděkování

Tato práce byla vypracována s podporou projektu Rozvoj lidských zdrojů ve výzkumu a vývoji moderních soft computingových metod a jejich praktického využití, reg. č. CZ.1.07/2.3.00/20.0072 podpořeného Operačním programem Vzdělávání pro konkurenceschopnost, financovaného ze strukturálních fondů EU a státního rozpočtu ČR.

Abstrakt

Analytické programování je jednou z metod symbolické regrese, při které skládáme základní jednoduché prvky do složitějších celků. Tento proces můžeme využít např. při aproximaci naměřených dat vhodnou matematickou formulí či syntéze logických obvodů. Pro práci analytického programování je zapotřebí evolučního algoritmu, speciální třídy optimalizačních algoritmů inspirované biologickými procesy. Těchto evolučních algoritmů bude popsáno a implementováno celkem pět. Cílem této práce tedy bude vytvořit algoritmus analytického programování v jazyce C# a otestovat jej na vybraných problémech. Součástí toho bude i implementace příslušných evolučních algoritmů a porovnání jejich výsledků. Dalším bodem srovnávání budou také dva použité generátory náhodných čísel.

Klíčová slova: optimalizace, evoluční algoritmy, evoluční strategie, simulované žíhání, rojení částic, diferenciální evoluce, SOMA, symbolická regrese, genetické programování, gramatická evoluce, analytické programování, metaevoluce, logistická rovnice, problém Quintic, problém Sextic, C#, .NET Framework

Abstract

Analytic programming is one of methods of symbolic regression in which we are composing simple elements into more complex units. This process can be used e.g. for approximation measured data by a suitable mathematical formula or for logical circuit synthesis. For analytic programming work is needed an evolutionary algorithm, special class of optimization algorithms which is inspired by biological processes. In this thesis there will be described and implemented five of them. The goal of this thesis is to create an algorithm of analytic programming in C# language and testing this algorithm on selected problems. Next part of this goal will be implementation of appropriate evolutionary algorithms and comparison of their results. Another point of comparison will be also two generators of random numbers.

Keywords: optimization, evolutionary algorithms, Evolution Strategies, Simulated Annealing, Particle Swarm Optimization, Differential Evolution, SOMA, symbolic regression, Genetic Programming, Grammatical Evolution, Analytic Programming, metaevolution, logistic equation, Quintic problem, Sextic problem, C#, .NET Framework

Seznam použitých zkratek a symbolů

AP	–	analytické programování
CFE	–	počet ohodnocení účelové funkce
DE	–	diferenciální evoluce
DSH	–	technika numerického manipulování s nenumerickými objekty
EA	–	evoluční algoritmus
ES	–	evoluční strategie
GE	–	gramatická evoluce
GFS	–	obecná funkční množina
GP	–	genetické programování
PSO	–	optimalizace rojením částic
SA	–	simulované žíhání
SOMA	–	samoorganizující se migrační algoritmus

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Analytické programování je jedním z přístupů k symbolické regresi, což je proces, při kterém např. prokládáme naměřená data vhodnou matematickou formulí. V tomto procesu v podstatě skládáme základní jednoduché prvky do složitějších celků. Takovéto celky nazýváme jednotně programy. Tyto programy nám mohou reprezentovat již zmíněnou matematickou formuli.

Naším úkolem je vytvořit matematickou formuli, která co nejlépe odpovídá prokládaným datům. Pro běh analytického programování je tedy zapotřebí optimalizačního algoritmu, který bude zjišťovat kvalitu vytvořených výrazů a pomocí analytického programování vytvářet čím dál tím kvalitnější výrazy. Takovéto algoritmy jsou často inspirovány přírodními procesy a označujeme je jako algoritmy evoluční.

Cílem této práce bude vytvoření algoritmu analytického programování v jazyce C# a jeho otestování na vybraných problémech. Pro jeho běh bude naprogramováno celkem pět evolučních algoritmů, jejichž výsledky budou porovnány. Pro běh evolučních algoritmů je mimo jiné zapotřebí generátoru náhodných čísel. K tomuto účelu nám poslouží Mersenne Twister a generátor náhodných čísel pomocí logistické rovnice. Výsledky průběhů evolucí pak budou pro oba dva generátory porovnány.

První část práce se bude zabývat problematikou evolučních algoritmů. Budou zde uvedeny základní pojmy a také zde bude popsán obecný průběh evolučních algoritmů. Pro účely této práce bylo vybráno pět evolučních algoritmů: evoluční strategie, simulované žíhání, rojení částic, diferenciální evoluce a samoorganizující se migrační algoritmus.

Druhá část práce pojednává o problému symbolické regrese a algoritmech pro symbolickou regresi. Jsou zde zmíněny tři algoritmy: genetické programování, gramatická evoluce a analytické programování. U analytického programování bude nastíněna technika posíleného hledání a technika generování konstant a jejich odhadování pomocí dalšího evolučního algoritmu.

Ve třetí části se budeme věnovat samotné implementaci analytického programování, evolučních algoritmů, generátorů náhodných čísel a také reprezentaci výrazů. Protože se jedná o techniku výpočetně velice náročnou, budeme při implementaci dbát na optimalizaci kódu.

Ve čtvrté části si popíšeme dva problémy, na kterých budeme testovat implementované evoluční algoritmy a generátory náhodných čísel. Tyto výsledky pak porovnáme.

2 Evoluční algoritmy

Většinu problémů inženýrské praxe lze definovat jako optimalizační úlohu. Jedná se např. o problémy nalezení optimální trajektorie robotu, nalezení optimální tloušťky stěny tlakové nádoby či nalezení optimálního tvaru křídla letadla. Řešení těchto problémů analytickou cestou je mnohdy nevhodné či nereálné. Do popředí tak nastupují tzv. optimalizační algoritmy.

V případě optimalizačních algoritmů je problém převeden na matematickou úlohu danou vhodným předpisem – funkcí. Řešením tohoto problému je pak optimalizace argumentů dané funkce. Z tohoto důvodu byla vytvořena třída algoritmů, jež se často inspirují přírodními procesy a obecně je označujeme jako evoluční algoritmy.

Podle strategie lze evoluční algoritmy rozdělit do dvou tříd:

- **Bodové** – metody založené na bodové strategii, jejich základem je operace sousedství aktuálního řešení, v němž hledáme řešení lepší.
- **Populační** – metody založené na strategii populace.

2.1 Základní pojmy z oblasti evolučních algoritmů

2.1.1 Účelová funkce

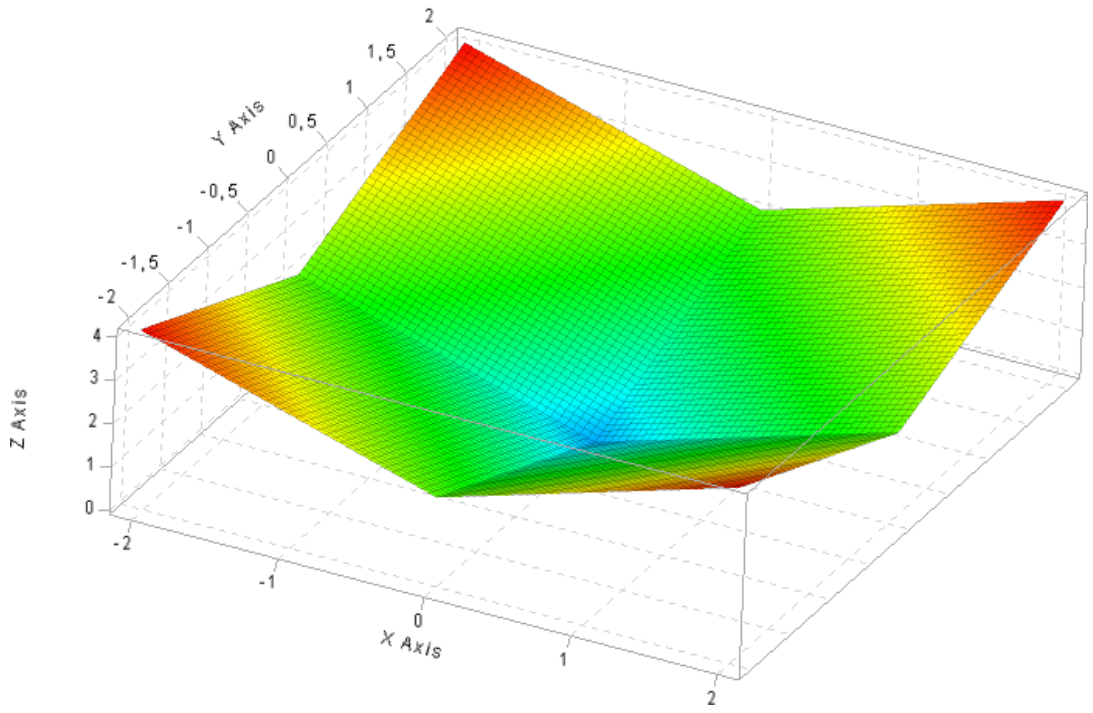
Účelová funkce $f(x)$ či $f_{cost}(x)$ je funkce, jejíž optimalizace (nalezení minima či maxima) vede k nalezení optimálních hodnot jejích argumentů. Na tuto funkci nahlížíme jako na geometrický problém, jehož optimum (minimum či maximum) hledáme na $N+1$-rozměrné ploše, kde N je počet argumentů optimalizované funkce. V této práci se dále budeme zabývat pouze hledáním minima účelových funkcí.

Funkce má v bodě x_0 lokální minimum, jestliže existuje okolí bodu x_0 takové, že platí $f(x_0) \leq f(x)$ pro všechna x z tohoto okolí. Funkce má v bodě x_0 ostré lokální minimum, jestliže existuje okolí bodu x_0 takové, že platí $f(x_0) < f(x)$ pro všechna x z tohoto okolí vyjma bodu $x = x_0$.

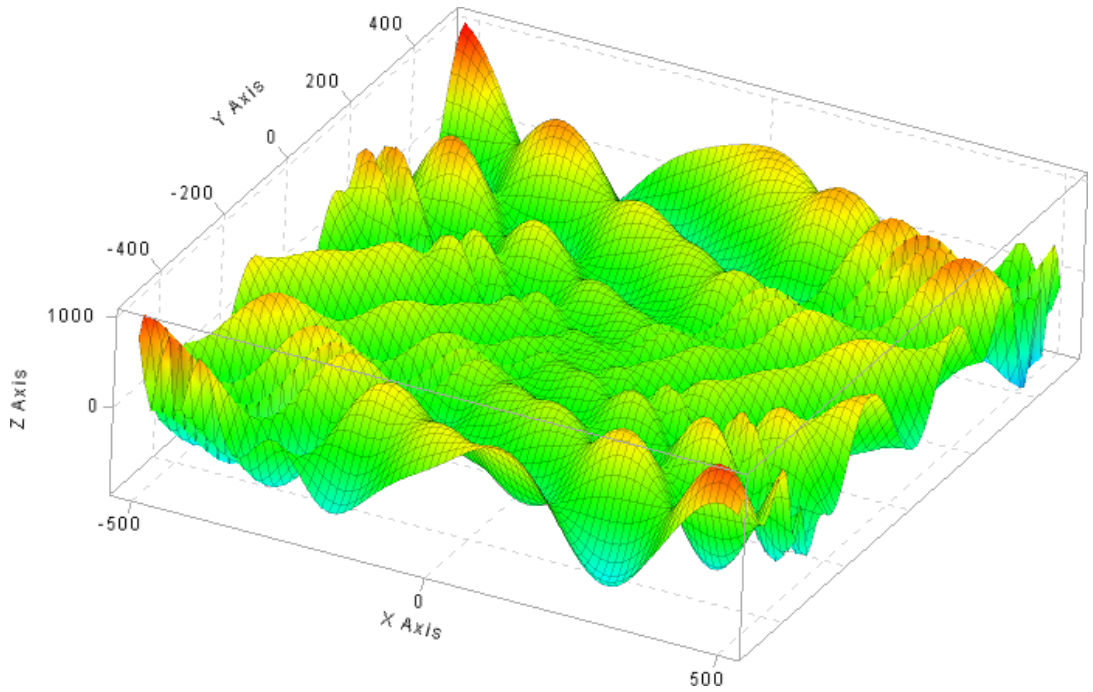
Je-li X nějaká neprázdná množina z euklidovského N-rozměrného prostoru E_N, pak funkce f má v bodě $x_0 \in X$ globální minimum vzhledem k X, jestliže se v x_0 nachází ostré lokální minimum pro všechna $x \in X$.

Tvorba účelové funkce je jedním z nejdůležitějších kroků v rámci optimalizačního procesu. Při její tvorbě musíme znát vše o daném problému, z čeho se má vycházet a čeho lze dosáhnout. Špatný návrh účelové funkce může negativně ovlivnit kvalitu výsledků.

Účelovou funkci, která má pouze jeden extrém, nazýváme unimodální. Naopak účelovou funkci s více extrémy nazýváme multimodální.



Obrázek 2.1: Unimodální účelová funkce



Obrázek 2.2: Multimodální účelová funkce

2.1.2 Jedinec

Jedincem rozumíme aktuální řešení daného problému. Jedná se v podstatě o množinu argumentů účelové funkce (bod v N-rozměrném prostoru), jejíž optimální číselná kombinace je hledána.

Dosazením daného jedince do účelové funkce a jejím vyhodnocením získáme její hodnotu – vhodnost neboli fitness – která říká, jak je daný jedinec vhodný pro další vývoj. Jedná se o kvalitu jedince.

Pro reprezentaci jedince můžeme použít několika způsobů [1]:

- **Binární** – jedinec je tvořen sekvencí jedniček a nul – chromozomem. Nevýhodou této reprezentace je skoková změna struktury chromozomů, kterou řeší použití Grayova kódu. Další nevýhodou binární reprezentace je přesnost – krátký jedinec může reprezentovat pouze číslo s málo místy za desetinnou čárkou. Naproti tomu rostoucí délka jedinců může přinést další problémy s operacemi v příslušných evolučních algoritmech.
- **Číselná** – jedinec je tvořen reálnými čísly nebo celými čísly, popř. jejich kombinací. Při použití jedince tvořeného reálnými čísly dosazovaného do účelové funkce s celočíselnými argumenty používáme dvě strategie:
  - Zaokrouhlení čísel přímo v jedinci
  - Zaokrouhlení čísel až před dosazením do účelové funkce – zvýšíme tak diverzitu a robustnost evolučního algoritmu
- **Nenumerická** – jedinec obsahuje nenumerické hodnoty (operátory, funkce apod.)
- **Strom** – umožňuje vizualizovat jedince jako stromovou strukturu

2.1.3 Vzorový jedinec

Vzorový jedinec neboli Specimen popisuje pro každého jedince typ (celé číslo, reálné číslo) a rozsah (dolní a horní mez) všech jeho argumentů [1]:

$$Specimen = \{\{Real, \{Lo, Hi\}\}, \{Integer, \{Lo, Hi\}\}, \ldots, \{Real, \{Lo, Hi\}\}\} \tag{2.1}$$

Tento vzorový jedinec je dále použit pro generování jedinců nových a kontrolu, zda jedinci vytvoření operacemi v evolučních algoritmech nezasahovali do zakázaných oblastí.

Volba hranic je důležitým krokem, protože v důsledku jejich nevhodného zvolení se může stát, že nalezená řešení budou nesmyslná nebo že je nebude možné fyzikálně realizovat.

2.1.4 Populace

Hlavním rysem evolučních algoritmů je práce s populací, tedy souborem jedinců. Jejich hlavní činností je cyklické vytváření nových populací jako náhrada za ty staré pomocí přesně definovaných matematických pravidel (v závislosti na použitém evolučním algoritmu).

Pro správný chod algoritmu je třeba nejdříve vygenerovat prvopočáteční populaci $P^{(0)}$ danou maticí $M \times N$, kde M je počet jedinců a N počet argumentů účelové funkce [1]:

$$P_{i,j}^{(0)} = x_{i,j}^{(0)} = rnd\,[0,1] \cdot \left(x_{i,j}^{(Hi)} - x_{i,j}^{(Lo)}\right) + x_{i,j}^{(Lo)} \qquad i = 1, \ldots, M; j = 1, \ldots, N \tag{2.2}$$

Pomocí tohoto vztahu zajistíme náhodné vygenerování všech parametrů jedinců uvnitř prostoru možných řešení.

2.1.5 Krizové stavy

U evolučních algoritmů se často stává, že vygenerovaná řešení (jedinci) zasahují i do zakázaných oblastí. Součástí evolučních algoritmů tak musí být i metody, které zajistí, že se nově vygenerovaná řešení budou nacházet ve vymezeném prostoru.

První z metod, jak udržet jedince ve vymezeném prostoru možných řešení, je nahrazení parametrů, které překročily hranici, novými a náhodně vygenerovanými parametry ležícími v povoleném intervalu. To má vliv na zvýšení diverzity populace, protože v okamžiku náhrady parametru je jedinec přesunut na novou pozici, která není výsledkem křížení jedinců.

Druhou z metod je záměrná úprava hodnoty účelové funkce jedinců, jejichž argumenty jsou neakceptovatelné – penalizace:

- **Hard-constraints** – jedinec nacházející se v zakázané oblasti je zrušen a nahrazen jedincem novým (vygenerovaným v povolené oblasti).
- **Soft-constraints** – jedinec, který leží v zakázané oblasti, je znevýhodněn modifikací hodnoty účelové funkce. Prostor možných řešení tak zůstává souvislý.

2.1.6 No Free Lunch teorém

No Free Lunch teorém nám říká, že neexistuje univerzální algoritmus, který by zvládl vyřešit všechny problémy lépe než algoritmy jiné. Evoluční algoritmy jsou testovány na rozsáhlém souboru testovacích funkcí a problémů a jejich výsledky jsou porovnávány. Určitě budou existovat problémy, které algoritmus A zvládne řešit lépe než algoritmus B a naopak.

2.2 Obecný průběh evolučních algoritmů

Existuje celá řada méně či více známých evolučních technik. V této části si popíšeme celkem pět evolučních algoritmů (budou popsány implementované verze algoritmů), které budou v dalších částech práce podrobeny testování. Nejdříve si však popíšeme obecný průběh evolučních algoritmů [1]:

1. **Definice parametrů algoritmu** – každý evoluční algoritmus musí mít definovány parametry, které řídí jeho běh nebo jej ukončí. Součástí definice parametrů je také definice účelové funkce. Mezi parametry, které mají následující evoluční algoritmy společné, patří:
   (a) **Dimenze problému** – neboli počet argumentů účelové funkce
   (b) **Velikost populace** – počet jedinců
   (c) **Počet iterací** – počet opakování vnitřních cyklů evolučního algoritmu (jiné pojmenování pro každý algoritmus, např. generace, migrace apod.)
2. **Vygenerování prvopočáteční populace** – podle parametrů definovaných výše je vygenerovaná počáteční populace jedinců, jejichž složky jsou nastaveny nahodile.
3. **Ohodnocení jedinců** – každý jedinec je ohodnocen pomocí účelové funkce a je mu přiřazena její hodnota. Tato hodnota je jedinci přidělena buď přímo, nebo je nejdříve transformována (normalizována) na tzv. fitness (vhodnost).
4. **Výběr rodičů** – rodiče jsou vybíráni na základě jejich kvality.
5. **Křížení** – křížením vybraných rodičů tvoříme potomky. Tento proces je u každého algoritmu jiný.
6. **Mutace** – potomek je pozměněn pomocí vhodného náhodného procesu.
7. **Ohodnocení jedinců** – každý potomek je ohodnocen viz bod 3.
8. **Výběr nejlepších jedinců**, jimiž bude následně zaplněna nová populace.
9. **Zapomenutí staré populace** a nahrazení populací novou. Dále pokračujeme krokem č. 4.

2.3 Evoluční strategie

Evoluční strategie (Evolution Strategies, ES) patří mezi první úspěšné stochastické[1] algoritmy v historii. Byl vyvinut na Technické univerzitě v Berlíně výzkumníky P. Bienertem, I. Rechtenbergem a H. P. Schwefelem [2].

Evoluční strategie můžeme značit způsoby buď $(\mu + \lambda)$-ES, nebo (μ, λ)-ES, kde parametry μ a λ značí velikost populace rodičů a potomků. Symboly „+“ a „,“ značí strategii výběru řešení do nové populace. Symbol „+“ znamená výběr jak rodičů, tak potomku, symbol „,“ nám říká, že do nové populace se mohou dostat pouze potomci.

[1] Algoritmy, jejichž kroky využívají náhodné operace. Výsledky řešení se tedy v jednotlivých běžích programu mohou lišit.

2.3.1 Parametry ES

Kvalita běhu ES je ovlivněna nastavitelnými parametry:

- **Dimension** – počet argumentů účelové funkce, dána problémem
- **Parents (μ)** – počet rodičů v populaci
- **Offspring (λ)** – počet potomků, kteří budou z populace rodičů vytvoření
- **Iterations** – maximální počet iterací ES
- **Deviation (σ)** – směrodatná odchylka pro Gaussovo normální rozdělení
- **Strategy** – strategie výběru řešení do nové populace
- **Recombination Parents (ρ)** – počet rodičů pro rekombinaci (viz níže)

2.3.2 Algoritmus ES

Základem algoritmu ES je jednoduchý iterativní proces, při kterém z rodičů vytváříme potomky pomocí tzv. Gaussova mutačního operátoru:

```
for cyklus < iterace do
begin
   y^i = N(x^i, σ) − tvorba λ nových potomků
   P = μ ∪ λ = (⋃_{j=1}^{λ} y^j) ∪ (⋃_{i=1}^{μ} x^i)
   μ = výběr μ nejlepších řešení z P
   if nejlepší f_cost(μ) < FV then
   begin
      Stop ES
   end
end
```

Výpis 1: Pseudokód ES

Veliký vliv na výkon má zde mutace – přičtení náhodně vygenerovaných čísel pomocí normálního rozdělení k rodiči. O této verzi se mluví jako o ES s elitismem, protože do nové populace jsou vybírání jak rodiče, tak potomci na základě jejich vhodnosti. Jelikož byla nalezena množina problémů, na nichž $(\mu + \lambda)$-ES stagnovala, byla navržena verze (μ, λ)-ES – ES bez elitismu. Pseudokód je totožný s pseudokódem uvedeným výše. Liší se pouze ve výběru jedinců do nové populace – vybíráno je pouze z potomků:

$$P = \bigcup_{j=1}^{\lambda} y^j \tag{2.3}$$

Zde ovšem musí platit, že $\lambda \geq \mu$.

2.3.3 Rekombinační ES

Rekombinační ES značíme $(\mu/\rho + \lambda)$-ES a od předchozí strategie se liší v tom, že je rodič pro mutaci (rekombinant) vytvořen rekombinací několika rodičů z populace. Počet rodičů pro rekombinaci je dán parametrem ρ. Hodnota parametru $\rho = 1$ znamená, že se rekombinace neuskutečňuje.

V současnosti existují dva druhy rekombinace: průměrová a diskrétní. Průměrová rekombinace je dána vztahem:

$$y_{rekombinant} = \frac{1}{\rho} \sum_{n=1}^{\rho} x^i \tag{2.4}$$

Diskrétní rekombinace je prováděna tak, že každý parametr rekombinanta je náhodně vybrán z ρ rodičů:

Rodič 1	$\boldsymbol{x_1^1}$	$\boldsymbol{x_2^1}$	x_3^1	x_4^1	x_5^1
Rodič 2	x_1^2	x_2^2	x_3^2	x_4^2	x_5^2
Rodič 3	x_1^3	x_2^3	x_3^3	x_4^3	x_5^3
Rodič 4	x_1^4	x_2^4	x_3^4	$\boldsymbol{x_4^4}$	x_5^4
Rodič 5	x_1^5	x_2^5	$\boldsymbol{x_3^5}$	x_4^5	$\boldsymbol{x_5^5}$
$\boldsymbol{y_{rekombinant}}$	x_1^1	x_2^1	x_3^5	x_4^4	x_5^5

Tabulka 2.1: Diskrétní rekombinace pro $\rho = 3$

Pseudokód strategie $(\mu/\rho + \lambda)$-ES je následovný (v případě strategie $(\mu/\rho, \lambda)$-ES je změna stejná jako u (μ, λ)-ES):

```
for cyklus < iterace do
begin
   Rekombinuj podle (2.4) nebo Tab. 2.1
   y^i = N(y^i_rekombinant, σ) − tvorba λ nových potomků
   P = μ ∪ λ = (⋃_{j=1}^{λ} y^j) ∪ (⋃_{i=1}^{μ} x^i)
   μ = výběr μ nejlepších řešení z P
   if nejlepší f_cost(μ) < FV then
   begin
      Stop ES
   end
end
```

Výpis 2: Pseudokód rekombinační ES

2.4 Simulované žíhání

Simulované žíhání (Simulated Annealing, SA) je optimalizační pravděpodobnostní metoda uvedena S. Kirkpatrickem [3], která hledá globální minimum funkcí majících několik lokálních minim. Inspirací pro tuto metodu bylo žíhání v metalurgii. Jedná se o proces, při kterém v kovu s nestabilní krystalovou mřížkou dochází ke stabilizaci volných částic k optimálnímu stavu tak, že daný kov zahřejeme na vysokou teplotu (k bodu tání) a velmi pomalu jej chladíme. Kov tak získá požadovanou optimální kvalitu.

2.4.1 Parametry SA

Kvalita běhu SA je ovlivněna nastavitelnými parametry:

- **Dimension** – počet argumentů účelové funkce, dána problémem
- **Population** – počet rodičů (stavů systému). Simulované žíhání je algoritmus založený na bodové strategii. Je však vhodné zvolit více výchozích stavů systému.
- **Neighbors** – počet sousedů každého rodiče (aktuálního řešení)
- **Deviation (σ)** – směrodatná odchylka pro Gaussovo normální rozdělení
- **TStart (T_0)** – počáteční teplota systému
- **TFinal (T_f)** – konečná teplota systému
- **Decr** – redukční faktor teploty
- **Repetitions (n_T)** – počet opakování Metropolisova algoritmu

Základem simulovaného žíhání je aplikace tzv. Metropolisova algoritmu, při kterém dochází opakovaným změnám systému z x_0 na x (je změněna poloha některé částice). Tento nový stav není automaticky akceptován, ale o jeho přijetí je rozhodnuto následujícím předpisem:

$$P\left(x \to x_0\right) = \begin{cases} 1 & \text{pro } f(x) < f(x_0) \\ e^{-\frac{f(x)-f(x_0)}{T}} & \text{pro } f(x) \geq f(x_0) \end{cases} \tag{2.5}$$

V případě, že nový stav x má nižší funkční hodnotu než předchozí, je přijat automaticky a nahradí stav starý, jinak je akceptován s pravděpodobností $0 < P\left(x \to x_0\right) < 1$. Při vysoké hodnotě teploty T se pak tato pravděpodobnost blíží jedné (akceptace většiny nových stavů), se snižující teplotou se však i tato pravděpodobnost snižuje (blíží se nule) a jen výjimečně je akceptován stav s vyšší funkční hodnotou.

Algoritmus simulovaného žíhání tedy opakuje Metropolisův algoritmus pro posloupnost klesajících teplot:

```
Náhodně vyber počáteční řešení x_0 z množiny všech přípustných řešení
x* je aproximací optimálního řešení
x* = x_0
repeat
   for i = 1 to n_T do
   begin
      Δf = f(x) − f(x_0)
      if Δf < 0 then
      begin
         x_0 = x
         if f(x) < f(x*) then
            x* = x
      end
      else
      begin
         Náhodně vyber r z rovnoměrného rozdělení na intervalu (0, 1)
         if r < e^(−Δf/T) then
            x_0 = x
      end
   end
   T = decr * T
until T > T_f
```

Výpis 3: Pseudokód SA

2.5 Rojení částic

Rojení částic (Particle Swarm Optimization, PSO) je populační optimalizační technika vyvinuta R. Eberhartem a J. Kennedym [4] inspirující se sociálním chováním živočichů, např. ptačích hejn. Pracuje na principu vytváření nových (lepších populací) následováním částic s lepší vhodností.

2.5.1 Parametry PSO

Kvalita běhu PSO je ovlivněna nastavitelnými parametry:

- **Dimension** – počet argumentů účelové funkce, dána problémem
- **Population** – počet částic v populaci
- **Migrations** – maximální počet migračních kol PSO
- **c1** – učící faktor ovlivňující tendenci posunu částic k nejlepšímu dosaženému řešení populace
- **c2** – učící faktor ovlivňující tendenci posunu částice ke svému dosavadně nejlepšímu nalezenému řešení

- **VMax** – maximální rychlost částic. Příliš nízká hodnota zapříčiní prohledávání malé oblasti. Naopak příliš vysoká hodnota způsobí přílišné vzdalování částic a překračování mezí prohledávané oblasti.
- **Setrvačnost (w)** – určuje setrvačnost pohybu částice. Malá hodnota podporuje hledání lokálních extrému, velká hodnota podporuje hledání extrémů globálních. Bývá tedy definovaná počáteční a koncová hodnota w_{start} a w_{end} a setrvačnost je během výpočtu v těchto mezích lineárně snižována. Zajistíme tím prohledávání funkce po velkých skocích na začátku algoritmu a prohledávání okolí nejlepšího jedince na konci algoritmu.

2.5.2 Algoritmus PSO

Populace je inicializovaná náhodně umístěnými částicemi, které mají svou rychlost a pamatují si svou dosud nejlepší pozici. Pozice částice s nejlepší hodnotou účelové funkce je uložena do společné paměti populace. Částicím je spočítán nový vektor rychlosti rovnice:

$$\begin{aligned} v_d(t+1) = w \cdot v_d(t) \\ + c_1 \cdot rand \cdot (pBest_{i,d} - x_{i,d}(t)) \\ + c_2 \cdot rand \cdot (gBest_{i,d} - x_{i,d}(t)) \end{aligned} \tag{2.6}$$

kde

$v_d(t+1)$	– rychlost jedince v následujícím kroku
$v_d(t)$	– rychlost jedince v aktuálním kroku
w	– setrvačnost jedince
$pBest_{i,d}$	– nejlepší dosavadní pozice daného jedince
$gBest_d$	– nejlepší nalezená pozice v populaci
$rand$	– náhodné číslo v intervalu $(0, 1)$
c_1, c_2	– učící faktory

Z rovnice lze usoudit, že částice se mohou ubírat třemi směry:

- **Individuální** – částice následují svou cestu
- **Konzervativní** – částice se vracejí na svou nejlepší dosaženou pozici
- **Přizpůsobivý** – částice následují nejlepšího jedince v populaci

Pokud rychlost částice přesáhne rychlost VMax, bude na tuto rychlost omezena. Následně je částice na základě vektoru rychlosti přesunuta na novou pozici:

$$x_{i,d}(t+1) = x_{i,d}(t) + v_d(t+1) \tag{2.7}$$

kde

$x_{i,d}(t+1)$ – pozice jedince v následujícím kroku

$x_{i,d}(t)$ – pozice jedince v aktuálním kroku

Částice, která se dostane za hranice prohledávaného prostoru, je nahrazena částicí novou a po přesunutí všech částic na nové pozice se celý cyklus opakuje. Tento proces můžeme ještě lépe znázornit v pseudokódu:

```
for i = 1 to Migrations do
begin
   for j = 1 to Population do
   begin
      Vypočítej rychlost částice podle (2.6)
      Uprav pozici částice podle (2.7)
      vhodnost = f_cost(p_j)
      if vhodnost < pBest_j then
         pBest_j = vhodnost
      if pBest_j < gBest then
         gBest = pBest_j
   end
end
```

Výpis 4: Pseudokód PSO

2.6 Diferenciální evoluce

Diferenciální evoluce (Differential Evolution, DE) je jedním z novějších algoritmů vyvinutý R. Stornem a K. Pricem [5]. Jedná se v podstatě o velice jednoduchý algoritmus, který však poskytuje velice dobré výsledky. Zvláštností tohoto algoritmu je provádění křížení až po mutačním procesu. Princip tvorby populace a ošetření parametrů jedinců je však stejný jako u ostatních evolučních algoritmů.

2.6.1 Parametry DE

Kvalita běhu DE je ovlivněna nastavitelnými parametry:

- **Dimension** – počet argumentů účelové funkce, dána problémem
- **Population** – počet jedinců v populaci
- **Generations** – maximální počet generací DE

- **CR** – jedná se o práh křížení. Volbou hodnotou tohoto parametru dáváme přednost ve výběru ze složek rodičů nebo šumových vektorů (viz níže). Hodnota CR by se měla pohybovat mezi 0 a 1, neměla by však těchto krajních hodnot nabývat. Při $CR = 0$ bude potomek kopií rodiče a evoluce probíhat nebude. Při $CR = 1$ bude potomek vytvořen pouze ze tří náhodně vybraných rodičů a algoritmus se tak bude podobat více náhodnému hledání než evoluci.

- **F** – mutační konstanta

2.6.2 Algoritmus DE

Princip činnosti algoritmu DE je velice jednoduchý. Pro každého jedince z populace náhodně vybereme tři další nestejné jedince r_1, r_2 a r_3, pomocí kterých vytvoříme tzv. šumový vektor podle následujícího vztahu:

$$v_j = x^G_{r_3,j} + F \cdot \left(x^G_{r_1,j} - x^G_{r_2,j}\right) \tag{2.8}$$

Máme-li spočtený šumový vektor, můžeme přistoupit ke křížení aktuálního jedince z populace a šumového vektoru – vytvoření zkušebního vektoru. Ten vytvoříme za pomocí prahu křížení CR tak, že postupně vybíráme parametry z aktuálního a šumového jedince a pro každou dvojici vygenerujeme náhodné číslo. Pokud je toto číslo menší než CR, do zkušebního vektoru přesuneme parametr z šumového vektoru a naopak.

Takto vytvořeného nového jedince pak ohodnotíme a jeho kvalitu porovnáme s rodičem. Do nové populace vložíme lepšího z nich. Tento proces můžeme ještě lépe znázornit v pseudokódu:

```
for i = 1 to Generations do
begin
   for j = 1 to Population do
   begin
      Náhodně vyber 3 nestejné jedince r_1, r_2 a r_3
      v = x_{r_3} + F · (x_{r_1} − x_{r_2}) // Tvorba šumového vektoru
      for k = 1 to Dimension do // Tvorba zkušebního vektoru
      begin
         Náhodně vyber r z rovnoměrného rozdělení na intervalu (0, 1)
         if r < CR then
            u_k = v_k
         else
            u_k = x_{j,k}
      end
      // Do nové populace přidáme lepšího z obou jedinců
      if f_cost(u) ≤ f_cost(x_{j,i}) then
         x_{j,i+1} = u
      else
         x_{j,i+1} = x_{j,i}
   end
end
```

Výpis 5: Pseudokód DE

2.7 Samoorganizující se migrační algoritmus

Samoorganizující se migrační algoritmus (Self Organizing Migrating Algorithm, SOMA) vyvinutý I. Zelinkou [6] je, podobně jako DE nebo PSO, algoritmus založený na vektorových operacích. Průběh algoritmu vychází z napodobení chování skupiny jedinců, kteří spolupracují při řešení společného problému, jako je např. hledání zdroje potravy.

2.7.1 Parametry SOMA

Kvalita běhu SOMA je ovlivněna nastavitelnými parametry:

- **Dimension** – počet argumentů účelové funkce, dána problémem
- **Population** – počet jedinců v populaci
- **Migrations** – maximální počet migrací SOMA
- **PathLength** – délka cesty určuje, jak daleko se aktivní jedinec zastaví od jedince, ke kterému migruje (k tzv. vedoucímu jedinci). Při $PathLength = 1$ se aktivní jedinec zastaví na místě vedoucího. Při $PathLength = 2$ se zastaví za vedoucím jedincem ve stejné vzdálenosti, v jaké startoval. Při $PathLength < 1$ se aktivní jedinec zastaví ještě před jedincem vedoucím, což má za následek degeneraci migračního procesu a nacházení pouze lokálních extrémů.
- **Step** – velikost jednoho kroku aktivního jedince směrem k vedoucímu jedinci. Tento parametr je důležité nastavit tak, aby vzdálenost mezi aktivním a vedoucím jedincem nebyla celočíselným násobkem parametru Step. Každý jedinec by tak mohl být přitažen jedincem vedoucím a algoritmus by mohl rychleji skončit v lokálním extrému.
- **PRT** – perturbace. Podle tohoto parametru budeme vytvářet tzv. perturbační vektor, pomocí kterého můžeme ovlivnit pohyb směrem k vedoucímu jedinci (zda se k němu bude aktivní jedinec pohybovat přímo či ne).
- **MinDiv** – minimální diverzita určuje, jaký je povolen minimální rozdíl mezi nejlepším a nejhorším jedincem v populaci. Je-li rozdíl menší než MinDiv, pak je algoritmus ukončen.

2.7.2 Algoritmus SOMA

Populace je inicializovaná náhodně umístěnými částicemi a následně je vybrán jedinec s nejlepší hodnotou účelové funkce. Dále každému jedinci před každým skokem vygenerujeme tzv. PRTVector podle vztahu:

$$PRTVector_j = \begin{cases} 1 & \text{pro } rnd_j < PRT \\ 0 & \text{jinak} \end{cases} \tag{2.9}$$

Pro každý parametr jedince je vygenerováno náhodné číslo v intervalu $\langle 0, 1 \rangle$. Je-li toto číslo menší než hodnota parametru PRT, pak je dané složce PRTVectoru přiřazena hodnota 1 a naopak:

$rand_j$	$PRTVector_j$
0,123	0
0,025	1
0,833	0

Tabulka 2.2: Příklad perturbačního vektoru pro $PRT = 0,1$ a jedince o 3 parametrech

Může se také stát, že všechny složky PRTVectoru budou rovny nule. V tom případě nastavíme libovolný parametr vektoru na 1. Novou pozici jedince pak spočteme pomocí následující rovnice:

$$\begin{gathered} x_{i,j}^{ML+1} = x_{i,j,start}^{ML} + \left(x_{L,j}^{ML} - x_{i,j,start}^{ML}\right) \cdot t \cdot PRTVector_j \\ t \in \langle 0, PathLength \rangle \end{gathered} \tag{2.10}$$

Takto provedeme posuv jedince po ploše pro všech $\frac{PathLength}{Step}$ kroků. Do nové populace nakonec přidáme jedince nacházejícího se na nejlepší pozici a celý proces opakujeme. Tento proces můžeme ještě lépe znázornit v pseudokódu:

```
for i = 1 to Migrations do
begin
   Vyber nejlepšího jedince – Leadera
   for j = 1 to Population do
   begin
      Vyber j–tého jedince
      for t = 0 to PathLength do
      begin
         Spočti novou pozici jedince podle (2.10)
      end
      Zapiš nejlepšího nalezeného jedince do nové populace
   end
   Vyber nejhoršího jedince – Worst
   if MinDiv > |f_cost(Leader) − f_cost(Worst)| then
      Stop SOMA
end
```

Výpis 6: Pseudokód SOMA

2.7.3 Strategie SOMA

Algoritmus SOMA byl vyvinut ve více variacích – strategiích:

1. **AllToOne** – všichni k jednomu. Tato strategie byla podrobně popsána výše. Všichni jedinci z populace migrují k jedinci nejlepšímu (vyjma jeho samotného).
2. **AllToOneRand** – všichni k jednomu náhodně. Funguje podobně jako strategie AllToOne s tím rozdílem, že jedinec, ke kterému všichni ostatní migrují, je vybrán z populace náhodně.
3. **AllToAll** – všichni ke všem. V této strategii migrují všichni jedinci ke všem ostatním podobně jako u strategie AllToOne. Po dokončení migrací aktuálního jedince se daný jedinec vrací na svou výchozí pozici. Tato strategie je výpočetně mnohem náročnější než AllToOne, ale je zde větší pravděpodobnost nalezení globálního extrému.
4. **AllToAllAdaptive** – všichni ke všem adaptivně. V této strategii, stejně jako v AllToAll, opět migrují všichni jedinci ke všem. Rozdíl je ovšem v tom, že po každé aktuálně dokončené migraci se jedinec přesune na svou dosavadně nejlepší pozici a z této pozice pak provádí migrace k jedincům dalším.

3 Symbolická regrese

Pojmem symbolická regrese označujeme proces, při kterém např. prokládáme naměřená data vhodnou matematickou formulí. V tomto procesu v podstatě skládáme základní jednoduché prvky do složitějších celků. Myšlenka, že bychom mohli řešit mnoho problémů pomocí symbolické regrese, napadla J. Kozu, který navrhl algoritmus genetického programování.

Symbolická regrese má široké využití. Mezi nejčastější možnosti aplikace bychom mohli zařadit:

1. Aproximace funkcí – prokládání naměřených dat vhodnou matematickou funkcí
2. Řešení diferenciálních rovnic
3. Syntéza logických obvodů
4. Syntéza neuronových sítí
5. Syntéza trajektorie robotů
6. Syntéza evolučních algoritmů

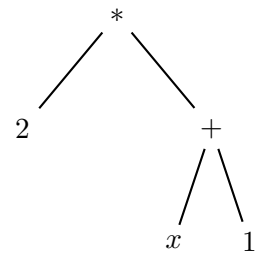
3.1 Genetické programování

Modifikací genetických algoritmů pro tvorbu tzv. programů (uživatelských programů, matematických výrazů, funkcí apod.) navrhl v 90. letech J. Koza [7] genetické programování (Genetic Programming, GP). Hlavní princip je tedy založen na genetických algoritmech a implementaci v jazyce LISP, který je schopný pracovat se symbolickými výrazy.

Jedinci, kteří se nacházejí v populaci, pak nejsou ve tvaru binárního řetězce, ale mohou se skládat z v podstatě libovolných objektů, např. $\sin$, $\cos$, $True$, $MoveRight$ apod. Z těchto objektů pak skládáme mnohem složitější výrazy, ve kterých hledáme ty nejvhodnější.

3.1.1 Struktura výrazu

Z dané množiny námi definovaných funkcí (např. $+$, $-$, $*$, $/$, x, 1, 2) lze vytvořit například výraz $2 \cdot (x + 1)$. Tento výraz je reprezentován tzv. syntaktickým stromem:

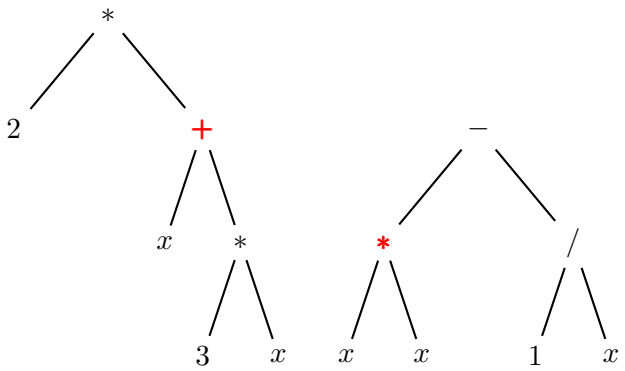


Obrázek 3.1: Syntaktický strom

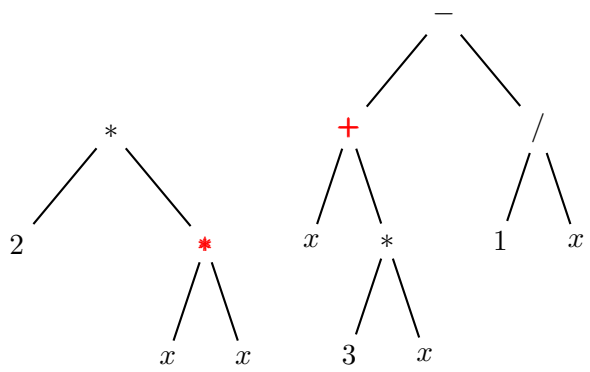
Kořenem stromu je zde operace na nejvyšší úrovni. Z každé operace vede tolik větví, kolik má daná operace argumentů a na každou větev znova aplikujeme přiřazení funkce. Tento proces provádíme tak dlouho, dokud nenarazíme na nerozložitelné výrazy – terminály (proměnné, čísla). Vyhodnocení výrazu pak probíhá od listů směrem ke kořenu.

3.1.2 Operace křížení

Podobně jako v genetických algoritmech (a v evolučních algoritmech obecně) i v genetickém programování provádíme operace křížení a mutace. V případě křížení operaci provádíme přímo jako modifikaci struktury stromu. U každého rodiče vybereme uzel křížení a podstromy těchto uzlů prohodíme. Vzniknou tak dva zcela nové programy (uzly křížení jsou barevně označeny):



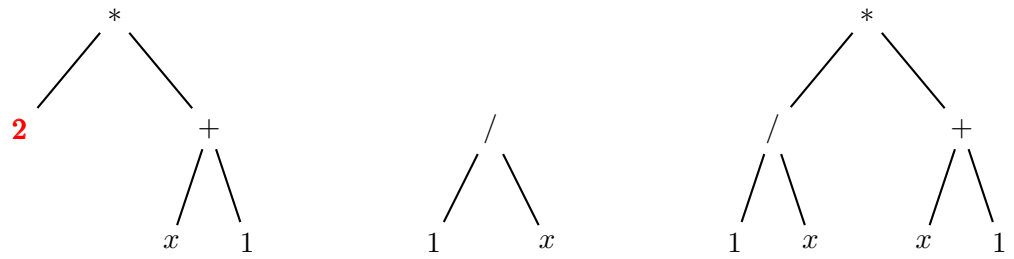
Obrázek 3.2: Křížení v genetickém programování – volba uzlů



Obrázek 3.3: Křížení v generickém programování – výsledek

3.1.3 Operace mutace

Další operací v pořadí je mutace potomka. Při mutaci opět vybereme uzel a jeho podstrom nahradíme náhodně vygenerovaným stromem:



Náhodně vygenerovaný strom

Obrázek 3.4: Mutace v genetickém programování

3.2 Gramatická evoluce

Genetické programování bylo původně programováno v jazyce LISP. Gramatická evoluce (Grammatical Evolution, GE), navržena M. O'Neillem a C. Ryanem [8], je v podstatě typ genetického programování založeného na gramatice. Můžeme tak vytvářet programy v libovolném jazyce v Backus-Naurově formě (BNF) za použití libovolného programovacího jazyka.

Gramatiky v BNF se skládají z terminálů ($1, x, +$, apod.) a neterminálů. Neterminály můžeme nahradit sekvencí terminálů a neterminálů. Takovou gramatiku pak můžeme zapsat jako čtveřici $G = \{N, T, P, S\}$ [9], kde

- **N** je konečná množina neterminálních symbolů
- **T** je konečná množina terminálních symbolů a platí $N \cap T = \emptyset$
- **S** je počáteční symbol a platí $S \in N$
- **P** reprezentuje množinu přepisovacích pravidel typu $A \to \beta$, kde
  - $\boldsymbol{A}$ je nonterminál, tedy $A \in N$
  - $\boldsymbol{\beta}$ je řetězec složený z terminálů a nonterminálů, tedy $B \in (N \cup T)^*$

Ukázkový příklad gramatiky generující aritmetické výrazy by mohl vypadat takto:

$$\begin{aligned} E &\to E + E | E \times E | (E) | N \\ N &\to DN | D \\ D &\to x | 0 | 1 | 2 | 3 \end{aligned} \tag{3.1}$$

3.2.1 Jedinec v GE

Samotný jedinec je v gramatické evoluci uložen ve formě binárního řetězce složený z osmibitových sekvencí – kodonů. Každý z těchto kodonů je následně převeden na celé číslo a pro výběr příslušného přepisovacího pravidla použijeme operaci modulo:

$$pravidlo = Integer\ hodnota\ kodonu\ MOD\ pocet\ moznosti\ aktualniho\ nonterminalu$$

Tímto způsobem postupně vybíráme přepisovací pravidla, dokud výsledný výraz neuzavřeme – výsledný výraz bude obsahovat pouze terminály. Následující příklad ukazuje proces sestavení programu za použití gramatiky uvedené výše. Mějme následující chromozom:

01111000	00100111	00100101	11011100	10011111	00000001	11000000
120	39	37	220	159	1	192

Tabulka 3.1: Příklad jedince pro gramatickou evoluci

Generování programu začne rozvinutím počátečního symbolu E. Pro tento symbol máme k dispozici celkem čtyři pravidla. Hodnota prvního kodonu je 120, a protože $120 mod 4 = 0$, počáteční symbol přepíšeme na $E + E$.

V dalších krocích budeme pokračovat vždy přepisováním nejlevějšího nonterminálu. V tomto případě je to opět E. Hodnota druhého kodonu je 39, $39 mod 4 = 3$, vybereme nonterminál N a dostáváme výraz $N + E$. Nyní budeme přepisovat nonterminál N, který obsahuje pouze 2 přepisovací pravidla. Hodnota následujícího kodonu je 37, $37 mod 2 = 1$, vybereme nonterminál D a získáme výraz $D + E$.

Nonterminál D obsahuje celkem 5 přepisovacích pravidel, hodnota následujícího kodonu je 220 a jelikož $220 mod 5 = 0$, dostáváme terminální symbol x a tím i výraz $x + E$. Tímto jsme uzavřeli první nonterminál. Obdobným způsobem budeme přepisovat a uzavřeme i druhý nonterminál a ve výsledku získáme výsledný výraz $x + 1$.

V ideálním případě (viz ukázkový příklad) přečteme celý kodon zleva doprava a po jeho přečtení dostaneme uzavřený výraz – výraz skládající se pouze z terminálů. Pokud dostaneme uzavřený výraz a v chromozomu zbývají nepoužité kodony, zbytek chromozomu ignorujeme. Pokud však přečteme celý chromozom a výraz ještě není uzavřen, chybějící kodony budeme číst znovu od začátku chromozomu. Při tomto postupu však může dojít k zacyklení, používáme proto omezení na počet průchodů chromozomem.

3.2.2 Operace křížení

V gramatické evoluci křížíme chromozomy standardním jednobodovým křížením. V místě křížení jsou chromozomy rozděleny na dvě části. První částí je kostra nového jedince a druhou jsou odříznuté větve. Křížení následně provedeme tak, že kostru nového jedince doplníme o uzly a podstromy, které budou generovány pomocí kodonů z druhé části druhého rodiče.

3.2.3 Operace mutace

Mutaci můžeme rozdělit na mutaci strukturálních genů a mutaci terminálních symbolů. Mutace strukturálních genů ovlivňuje geny ostatní (hrozí poškození nalezené struktury), proto by měla být velice malá – nové struktury můžeme vytvářet křížením. V případě mutace terminálních symbolů mutujeme pouze terminální symboly, struktura výrazu zůstává pořád stejná. Například z funkce $x + \sin(x + 3)$ tak můžeme pouze pomocí mutace vygenerovat funkci $2 \cdot \tan(1 - x)$.

3.3 Analytické programování

Jedním z novějších přístupů k symbolické regresi je analytické programování (Analytic Programming, AP) navržené I. Zelinkou [10]. Analytické programování není, na rozdíl od genetického programování či gramatické evoluce, propojeno s žádnou stromovou strukturou nebo gramatikou a neváže se pouze k jednomu algoritmu. AP může pro svůj chod využívat jakýkoliv evoluční algoritmus.

3.3.1 Objekty v AP

AP je postaveno na množině funkcí, operátorů a terminálů, stejně jako GP či GE:

- **Funkce**: $\sin$, $\cos$, $\tan$, $\log$, and, or, ...
- **Operátory**: $+$, $-$, $*$, $/$, ...
- **Terminály**: x, y, 1, 2, π, 4, ...

Tyto objekty jsou seřazeny podle počtu svých argumentů do obecné funkční množiny (General Functional Set, GFS), která je tvořena hierarchicky uspořádanými podmnožinami funkcí obsahující stejný počet argumentů. Tyto podmnožiny značíme následovně:

- $\boldsymbol{GFS_{all}}$: všechny elementy v GFS (sjednocením všech $GFS_{?arg}$)
- $\boldsymbol{GFS_{0arg}}$: funkce s 0 argumenty – terminály
- $\boldsymbol{GFS_{1arg}}$: funkce s 1 argumentem atd.

3.3.2 Princip činnosti AP

Činnost AP je velice jednoduchá. Jako vstup nám poslouží celočíselný jedinec z příslušného evolučního algoritmu, jehož jednotlivé složky ukazují do množiny základních symbolických objektů (tuto techniku numerického manipulování s nenumerickými objekty nazýváme DSH – Discrete Set Handling). Mějme tedy následující GFS:

$$GFS_{all} = \{+, -, *, \sin, \cos, x, 1, 2, \pi\}$$

Dále mějme celočíselného jedince:

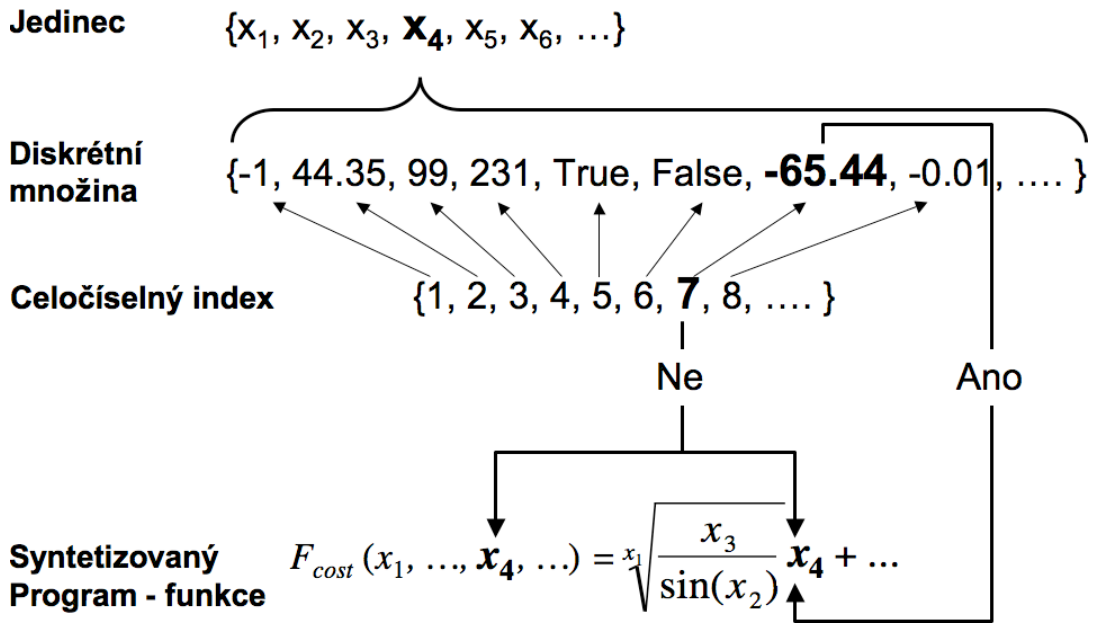
$$Jedinec = \{3, 8, 4, 16, 9\}$$

Generování výrazu začne výběrem prvního parametru jedince, čísla 3. Z GFS tedy vybereme třetí funkci $*$. Operátor násobení má dva argumenty, budeme tedy potřebovat další dva parametry jedince – 8 a 4, které ukazují na terminál 2 a funkci $\sin$. Tyto dvě funkce dosadíme jako argumenty do operátoru $*$, čímž získáme výraz $2 * \sin(?)$.

Terminál 2 má nulový počet argumentů, přesuneme se k poslednímu nepřiřazenému argumentu – argumentu funkce $\sin$. Další parametr jedince je číslo 1, které ukazuje na operátor $+$. Ten dosadíme do naší funkce a získáme výraz $2 * \sin(?+?)$.

Jak vidíme, ve výrazu se nám objevily dva nevyužité argumenty. Použijeme proto další dva parametry jedince 6 a 9, které ukazují na terminály x a π. Tyto terminály dosadíme na příslušné pozice ve výrazu a získáme tím výsledný výraz $2 * \sin(x + \pi)$.

V tomto výrazu již nejsou žádné nepřiřazené argumenty, výraz je uzavřen a přečetli jsme celého jedince. Skládání výrazu je hotovo. Příklad skládání výrazu si můžeme názorně ukázat na následujícím obrázku:



Obrázek 3.5: Princip práce s diskrétní množinou (převzato z [1])

Podobně jako v GE může dojít (a zcela jistě dojde) k tomu, že bude výraz uzavřen ještě před tím, než přečteme celého jedince. Zbytek jedince pak ignorujeme. Opačným případem je ten, kdy není výraz po přečtení jedince uzavřen. V tomto případě měříme vzdálenost od konce celočíselného jedince a podle této vzdálenosti pak určujeme, z jaké podmnožiny GFS budou vybírány příslušné funkce. Toto měření vzdálenosti během syntézy tak vždy zaručuje vygenerování nepatologického programu.

3.3.3 Příklady výrazů vygenerovaných v AP

$$(\tan(\sin(\pi+\pi)) - \cos(\tan(x+x))) \cdot (x - (\sin(\pi) \cdot \sin(\sin(\pi)))) \tag{3.2}$$

$$\left(\frac{\frac{\pi}{x}}{\pi+x} + \sin(\pi) - \cos(x)\right) + x \tag{3.3}$$

$$\cos\left(\frac{\pi}{\pi} + \left(\pi - \frac{\sin(\pi)}{\sin(\cos(\pi))}\right)\right) - \frac{\frac{x\cdot\tan(\pi)}{\tan(\cos(x))} + \cos(\tan(x))}{\sin(\sin(\sin(x) - \cos(\tan(\sin(x)))))} \tag{3.4}$$

3.3.4 Operace křížení a mutace

Operace křížení a mutace jsou plně v režii evolučního algoritmu, na který je analytické programování napojeno. AP tedy v podstatě funguje jako zobrazení množiny vybraných objektů do množiny programů.

3.3.5 Konstanty v AP

Práce s konstantami v AP byla vyzkoušena ve třech verzích. Všechny tři verze používají stejnou množinu funkčních a terminálních symbolů. První verze označovaná jako AP_{basic} pracovala s konstantami stejně jako GP. Do GFS bylo společně s funkcemi a operátory také přidáno velké množství náhodně vygenerovaných konstant. V tomto případě však citelně narostl počet všech možných kombinací funkcí.

Proto byly vyvinuty další verze AP: AP_{meta} a AP_{nf}. V těchto verzích je do GFS přidaná pouze jedna obecná konstanta K, která je před ohodnocením programu oindexována jako K_1, K_2, ..., K_n a tyto konstanty jsou následně numericky odhadnuty. Námi výše vygenerovaný výraz $2 * sin(x + \pi)$ by v tomto případě mohl vypadat takto:

$$K_1 \cdot \sin(x + K_2) \tag{3.5}$$

Ve verzi AP_{meta}, která bude v této práci implementovaná, odhadujeme hodnoty konstant za pomocí dalšího evolučního algoritmu. Struktura celého procesu je následující: $EA_{master} \rightarrow$ syntéza programu v AP $\rightarrow$ indexace konstant $\rightarrow$ spuštění $EA_{slave} \rightarrow$ nastavení konstant. Tato verze je tedy nazývána AP s metaevolucí. Podřízený proces EA_{slave} je spouštěn pro odhad konstant zvlášť u každého programu, což je časově velice náročné.

Poslední verzí AP je AP_{nf}, která pro odhad konstant používá numerickou metodu nelineárního prokládání (Nonlinear Fitting, NF) [11].

3.3.6 Posílené hledání

Do AP byla kvůli málo uspokojivým výsledkům zavedena technika tzv. posíleného hledání (Reinforced Search). Tato technika spočívá v přidání aktuálně syntetizovaného programu, jehož vhodnost je pod námi definovanou hranicí, do množiny GFS_{0arg}. S tímto programem dále pracujeme jako s nezávisle proměnnou.

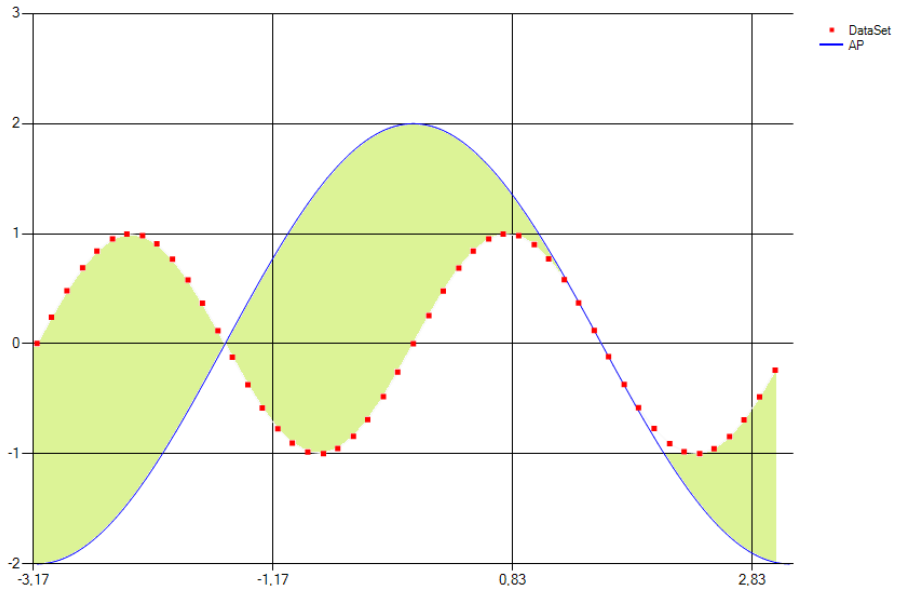
Pokud během dalšího hledání nalezneme program s vhodností ještě lepší, než je vhodnost programu do GFS_{0arg} již zavedeného, pak je předchozí program nahrazen programem novým. Tímto způsobem nezvyšujeme kardinalitu GFS, ale zvyšujeme kvalitu této množiny.

3.3.7 Vhodnost jedince v AP

Pomocí analytického programování transformujeme celočíselného jedince na určitý program. Další částí je ohodnocení tohoto jedince. V tomto případě bude vhodnost jedince rovna kvalitě syntetizovaného programu (nyní pro jednoduchost uvažujme pouze matematický výraz).

Kvalita matematického výrazu je rovna ploše mezi prokládanými daty a syntetizovanou funkcí. V praxi pak vezmeme hodnotu každého bodu prokládaných dat a funkční hodnotu syntetizované funkce v tomto bodě. Výsledná hodnota účelové funkce je pak dána součtem absolutních hodnot rozdílů mezi funkcemi ve všech daných bodech [1]:

$$f_{cost} = |DataSet - f_{AP}| \tag{3.6}$$



Obrázek 3.6: Rozdíl mezi prokládanými daty a syntetizovanou funkcí

4 Implementace

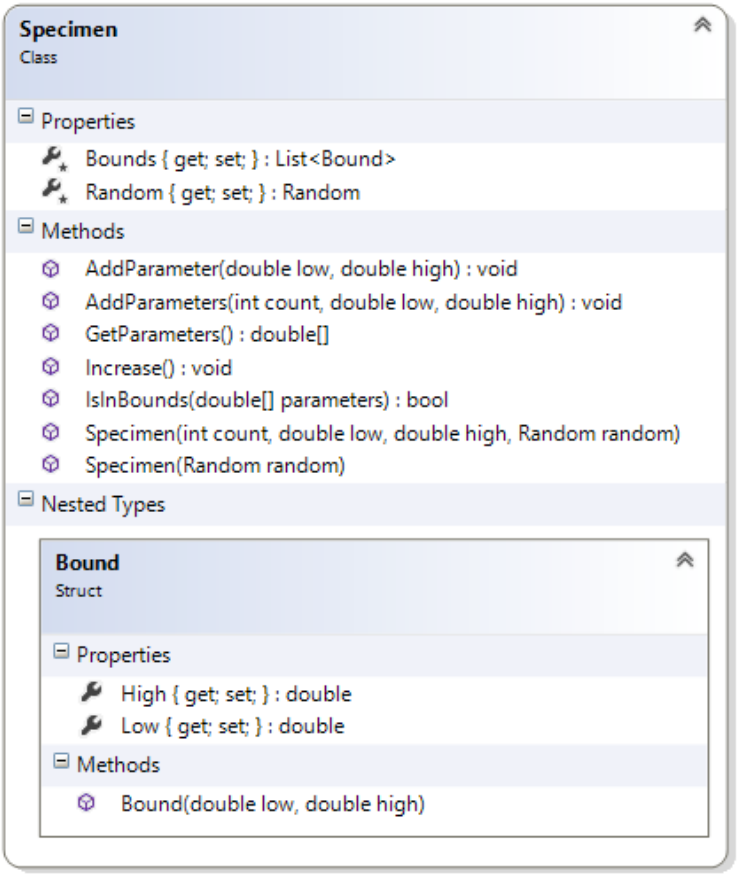
Implementace analytického programování bude realizována jako konzolová aplikace v jazyce C# a bude využívat .NET Framework verze 4.5. Jedinou výjimkou bude použití generátoru pseudonáhodných čísel Mersenne Twister [12].

Protože v této práci budou implementovány jen některé evoluční algoritmy a analytické programování bude použito zatím jen pro aproximaci funkcí, bylo by velice vhodné implementovat projekt tak, aby šel v budoucnu jednoduše upravit či rozšířit. Bude proto rozdělen na více částí.

Výsledný projekt je složen z 33 souborů (struktury, třídy, interface) a ty nejdůležitější z nich si nyní popíšeme.

4.1 Evoluční algoritmy

4.1.1 Třída Specimen



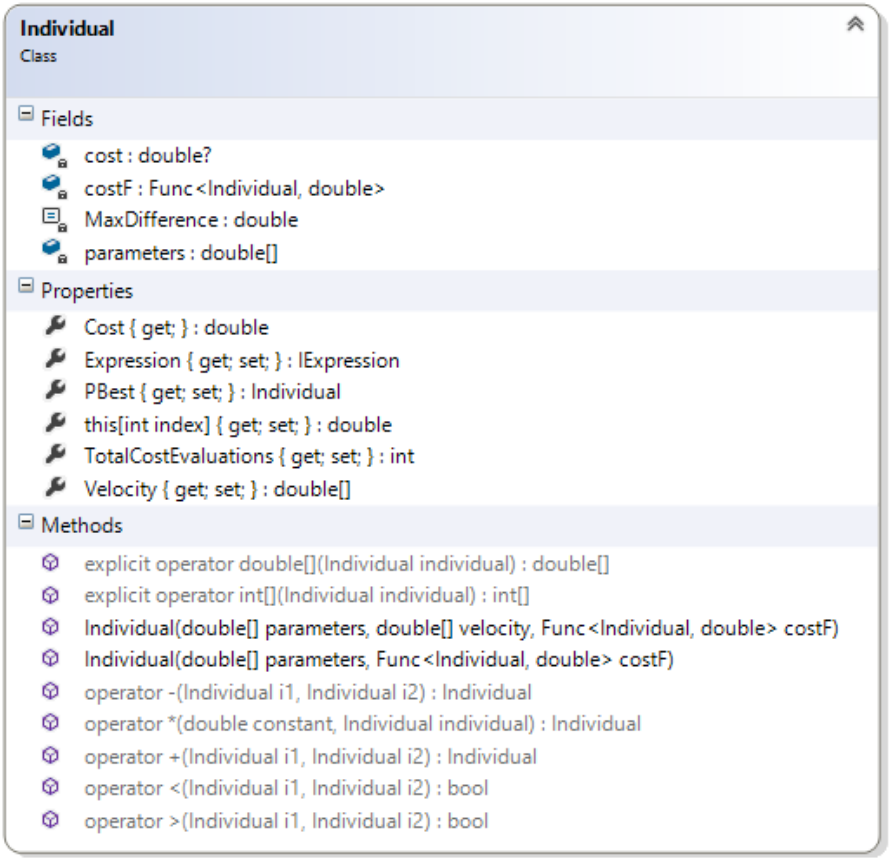
Obrázek 4.1: Třída Specimen

Třídou `Specimen` definujeme vzorového jedince – povolená oblast, ve které se jedinci mohou nacházet a do které budou generováni. Můžeme vytvořit buď prázdného vzorového jedince, ve kterém můžeme postupně definovat rozsah všech jeho parametrů, nebo můžeme rovnou vytvořit vzorového jedince s určitým počtem parametrů ve stejném rozsahu.

Při vytváření nových vektorů jedinců můžeme jejich platnost zkontrolovat pomocí metody `IsInBounds`, popř. vygenerovat zcela nový vektor jedince metodou `GetParameters`.

Pro účely posíleného hledání analytického programování je zde metoda `Increase`, která zvýší horní mez každého parametru vzorového jedince o 1.

4.1.2 Třída Individual



Obrázek 4.2: Třída Individual

Jedinec je zde reprezentován jako pole reálných hodnot. Jedinci definujeme také účelovou funkci, popř. vektor rychlosti (pro účely PSO). Může být indexován dle parametrů, pamatuje si svou dosavadní nejlepší pozici také výraz, který mu náleží.

Pro vektorové operace s jedinci zde máme připravené sčítání, odčítání a násobení konstantou. V praxi tak nemusíme sami procházet celého jedince, ale můžeme použít operátoru:

```
Individual ind1 = population[1];
Individual ind2 = population[2];
Individual ind3 = population[3];
Individual noisy = ind3 + parameters.F * (ind1 - ind2);
```

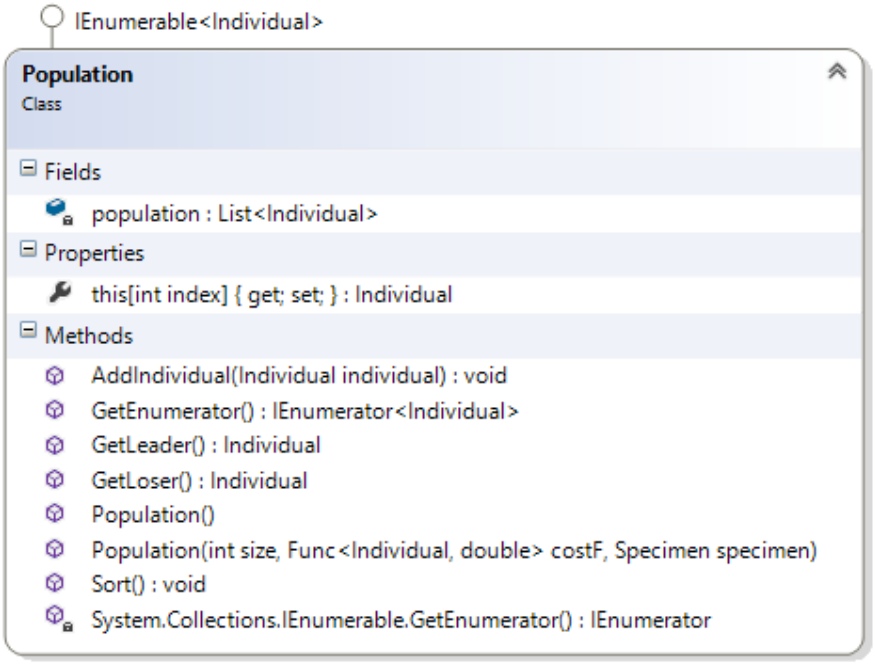
Výpis 7: Příklad použití přetížených operátorů

Pro ohodnocování jedince je použita velice časově náročná účelová funkce (probíhá zde další evoluce konstant – AP_{meta}). Její je proto uložena v proměnné typu `double?` a je počítána pouze jednou. Ve výsledku nám tento přístup ušetří spoustu času.

Pro porovnávání jedinců jsou zde přetíženy operátory $<$ a $>$, které porovnávají hodnoty účelových funkcí jedinců. Pokud jsou si hodnoty účelových funkcí velmi blízké, je jako lepší vybrán jedinec s kratším výrazem (důvod viz posílené hledání níže).

Nakonec jsou přetíženy operátory pro převod jedince na pole reálných nebo celých čísel. Převod jedince na pole celých čísel je realizován zaokrouhlením čísel reálných, ze kterých se jedinec skládá.

4.1.3 Třída Population

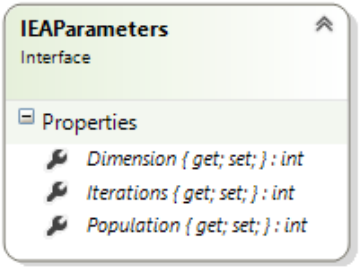


Obrázek 4.3: Třída Population

Populací rozumíme soubor jedinců. Populaci můžeme založit buď prázdnou a postupně do ní přidávat jedince pomocí metody `AddIndividual`, nebo vygenerovat populaci o určitém počtu jedinců na základě jedince vzorového.

Pro účely výběru těch nejlepších (např. evoluční strategie) máme možnost populaci setřídit podle hodnoty účelové funkce. Můžeme také vybrat nejlepšího nebo nejhoršího jedince z populace.

4.1.4 Rozhraní IEAParameters



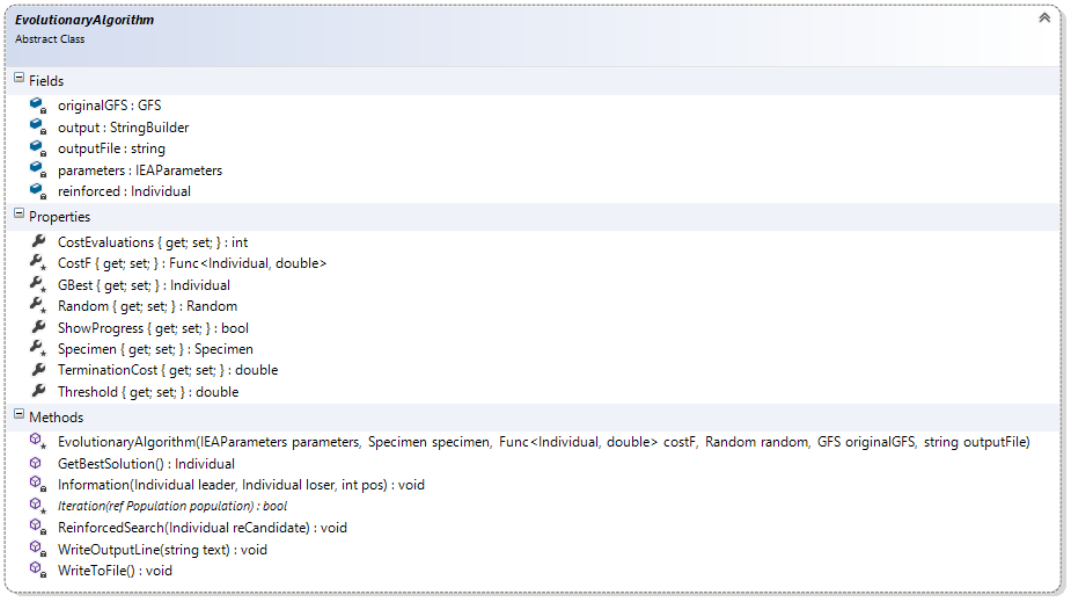
Obrázek 4.4: Rozhraní IEAParameters

Toto rozhraní nám definuje tři základní vlastnosti každého evolučního algoritmu:

- Dimenze
- Počet jedinců v populaci
- Počet iterací evolučního algoritmu

Na základě tohoto rozhraní pak budeme definovat parametry pro každý námi implementovaný evoluční algoritmus.

4.1.5 Třída EvolutionaryAlgorithm



Obrázek 4.5: Třída EvolutionaryAlgorithm

Tato třída abstrahuje základní funkci každého následně implementovaného evolučního algoritmu. Ten bude předávat této třídě své parametry, vzorového jedince, účelovou funkci a generátor náhodných čísel. Pro účely analytického programování je nutné zadat množinu funkcí a pro výpis informací o průběhu algoritmu ještě cestu k výstupnímu souboru.

Metodou, kterou celý algoritmus spustíme, je `GetBestSolution`, která cyklicky volá metodu `Iteration` a nakonec vrátí nejlepší nalezené řešení během evoluce. Metoda `Iteration` je abstraktní a je to v podstatě jediná věc (většinou), kterou musíme při psaní nového evolučního algoritmu vytvářet.

Pro účely posíleného hledání je zde metoda `ReinforcedSearch`, která je volána v každé iteraci algoritmu. Ta funguje přesně tak, jak jsme si ji popsali výše. Bohužel zde ale (podobně jako u genetického programování) dochází k tzv. Bloat efektu. Při tom postupně narůstá velikost výrazu – do množiny terminálů je přidán syntetizovaný program, který je pak použit v nových programech a eventuálně zase zařazen mezi terminály. Z tohoto důvodu do množiny terminálů nebudeme přidávat výrazy, jejichž strom má přes 100 000 uzlů.

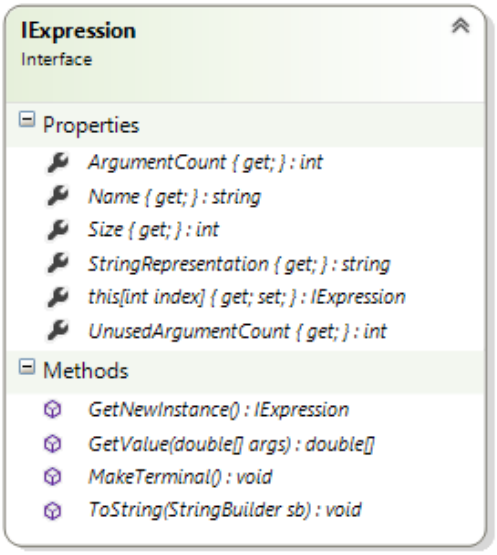
```
private void ReinforcedSearch(Individual reCandidate)
{
    if (originalGFS != null && reCandidate.Cost < Threshold
                            && reCandidate.Expression.Size < 100000)
    {
        IExpression expr = reCandidate.Expression.GetNewInstance();
        expr.MakeTerminal();
        if (reinforced == null)
        {
            originalGFS.AddFunction(expr); // Poprvé je výraz přidán do GFS
            Specimen.Increase();
        }
        else
        {
            originalGFS[originalGFS.Count - 1] = expr; // Poslední výraz v GFS je nahrazen novým
        }
        reinforced = reCandidate;
        Threshold = reinforced.Cost;
    }
}

public virtual Individual GetBestSolution()
{
    Population population = new Population(parameters.Population, CostF, Specimen);
    int i;
    for (i = 0; i < parameters.Iterations; i++)
    {
        Individual leader = population.GetLeader();
        Individual loser = population.GetLoser();
        Information(leader, loser, i); // Výpis informací o průběhu
        if (leader.Cost < TerminationCost)
        {
            i++;
            break;
        }
        ReinforcedSearch(leader);
        if (! Iteration (ref population))
        {
            i++;
            break;
        }
    }
    Individual popBest = population.GetLeader();
    Individual popWorst = population.GetLoser();
    Individual result = GBest != null && GBest < popBest ? GBest : popBest;
    Information(popBest, popWorst, i);
    WriteToFile();
    return result;
}
```

Výpis 8: Hlavní funkce bázové třídy EvolutionaryAlgorithm

4.2 Reprezentace výrazů

4.2.1 Rozhraní IExpression



Obrázek 4.6: Rozhraní IExpression

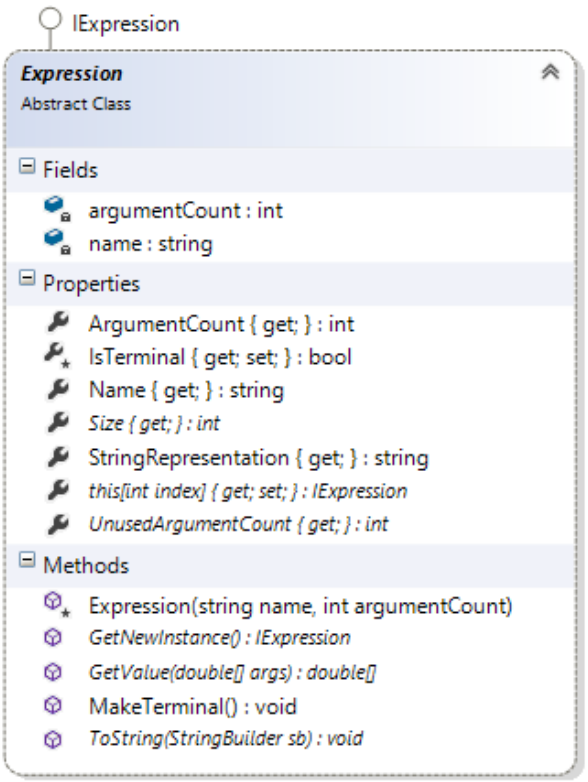
Rozhraní `IExpression` je základem všech výrazů. Každý výraz v sobě nese informaci o počtu svých argumentů, svém názvu, velikosti (počet uzlů stromu reprezentujícího tento výraz), jeho výsledné textové reprezentaci a počtu dosud nepřiřazených argumentů (v celém podstromu). Z každého výrazu je také možné dostat libovolný jeho argument, který je opět typu `IExpression`.

Jelikož budou výrazy odvozené od tohoto rozhraní většinou referenčního typu (třídy) a při prostém přiřazení jednoho výrazu jako argumentu jiným výrazům by došlo k jejich kolizi (při změně argumentů jednoho výrazu by došlo ke změně argumentů výrazu druhého), budeme při přiřazování výrazů do argumentů používat metodu `GetNewInstance`. Ta vytvoří přesnou kopii struktury původního výrazu.

Hodnotu funkce v daném bodě zjistíme tak, že do ní bod dosadíme. V případě rozhraní `IExpression` však nebudeme dosazovat pouze jeden bod, ale všechny body prokládaných dat najednou. Budeme tak procházet mnohdy velké stromy výrazů pouze jednou.

Pokud budeme vytvořený výraz přidávat to GFS v rámci posíleného hledání, označíme ho jako terminál. Tím se změní počet jeho argumentů na 0 a budeme s ním pracovat jako s nezávisle proměnnou. Při jeho použití ve výrazech pak bude při dosazovaní hodnot do metody `GetValue` přímo vrácen uložený výsledek bez dalšího počítání a procházení stromu – výraz už totiž nemůže být změněn.

4.2.2 Třída Expression

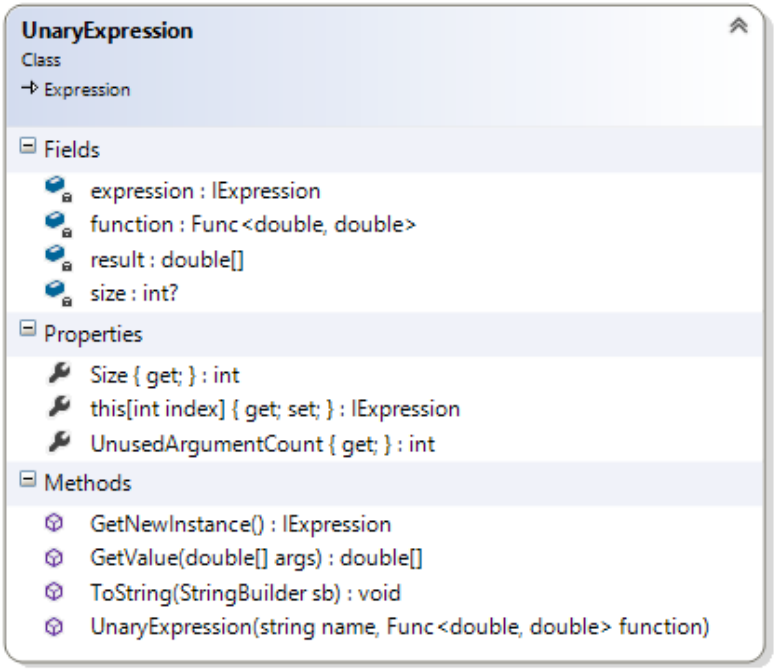


Obrázek 4.7: Třída Expression

Abstraktní třída `Expression` implementuje všechny prvky, které budou následně odvozené výrazy společné a nemusí je tak implementovat znova a znova. Jedná se o název, počet argumentů, funkce `MakeTerminal` a vlastnost `StringRepresentation`, která vrátí výraz ve formě řetězce.

Třídě `Expression` je také přiřazen atribut `DebuggerDisplayAttribute`, díky kterému v Debug módu místo názvu typu zobrazí výraz, což nesmírně usnadní práci při ladění.

4.2.3 Třída UnaryExpression



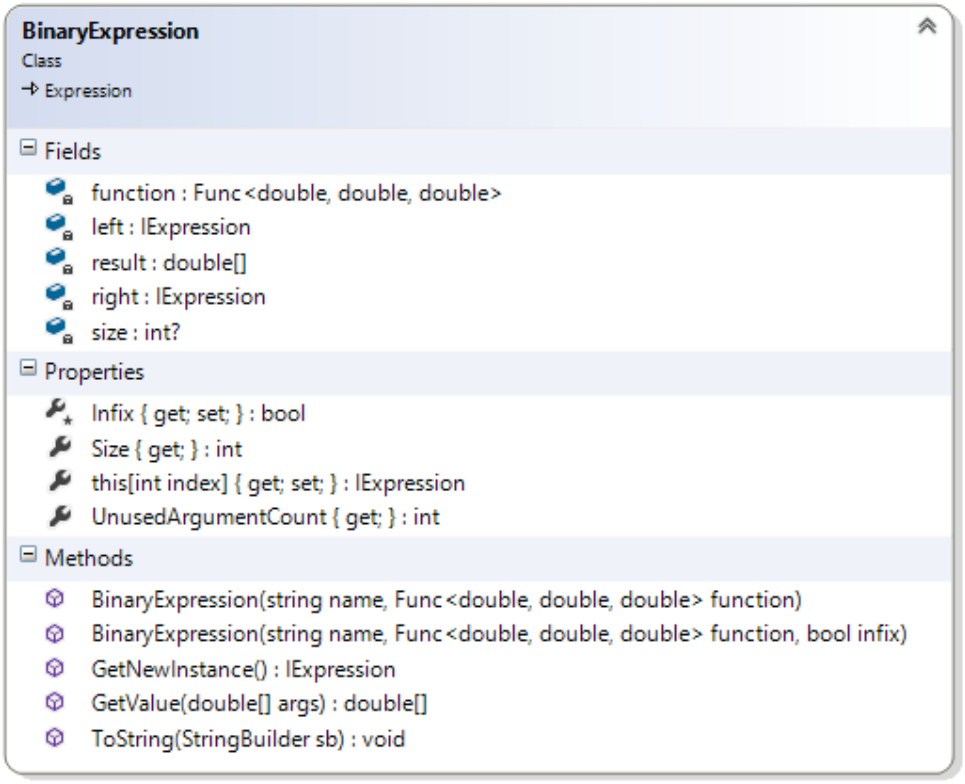
Obrázek 4.8: Třída UnaryExpression

Tato třída reprezentuje unární výrazy – funkce s jedním argumentem (např. sin, cos, tan, apod.) a implementuje všechny zbývající abstraktní metody třídy `Expression`. Při definování nového výrazu nám stačí zadat pouze název a funkci reprezentující tento výraz:

```
public static IExpression Sin
{
    get
    {
        return new UnaryExpression("sin", x => Math.Sin(x));
    }
}
```

Výpis 9: Ukázka vytvoření unárního výrazu

4.2.4 Třída BinaryExpression



Obrázek 4.9: Třída BinaryExpression

Tato třída reprezentuje binární výrazy – operátory nebo funkce se dvěma argumenty (např. $+$, $-$, $power$, apod.) a implementuje všechny zbývající abstraktní metody třídy `Expression`. Při definování nového výrazu nám stačí zadat pouze název a funkci reprezentující tento výraz:

```
public static IExpression Add
{
    get
    {
        return new BinaryExpression("+", (a, b) => a + b);
    }
}
```

Výpis 10: Ukázka vytvoření binárního výrazu

Při vytváření nového výrazu můžeme definovat, zda bude funkce vypisována v infixovém nebo v prefixovém tvaru.

4.2.5 Proměnná a konstanta

Proměnná a konstanta (třídy `X` a `Constant`) jsou terminální symboly, které jsou odvozeny přímo z rozhraní `IExpression`. Konstanta navíc definuje vlastnost `Value`, která reprezentuje její hodnotu.

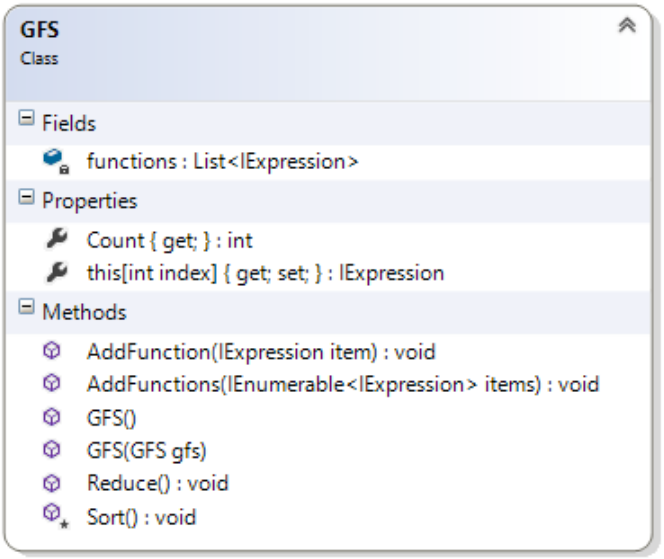
4.2.6 Předdefinované funkce

Pro pozdější využití byly nadefinovány i nejčastěji používané funkce:

- **Binární**: $+$, $-$, $*$, $/$, $power$
- **Unární**: $\sin$, $\cos$, $\tan$

4.3 Analytické programování

4.3.1 Třída GFS



Obrázek 4.10: Třída GFS

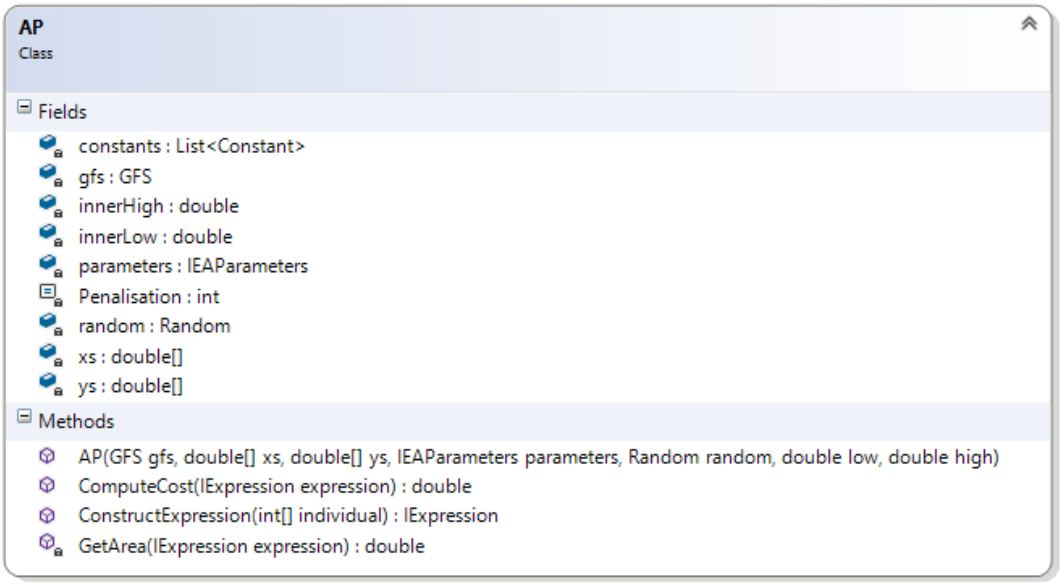
GFS je složena z pole prvků typu `IExpression`. Vytvořit můžeme buď prázdnou GFS a následně do ní přidat funkce, nebo vytvořit kopii existující GFS. Po přidání funkce do GFS jsou funkce v ní obsažené setříděny dle počtu argumentů.

Funkcí `Reduce` z GFS odstraníme všechny funkce s nejvyšším počtem argumentů (postupné přecházení do podmnožin GFS obsahující funkce s menším počtem argumentů).

Při volbě funkce na určité pozici je vrácena kopie uložené funkce. Z důvodu postupného přecházení do podmnožin GFS jsou funkce vybírány pomocí operace modulo:

$$funkce = index \; MOD \; |GFS|$$

4.3.2 Třída AP



Obrázek 4.11: Třída AP

Nejdůležitější třída celého projektu, která transformuje jedince na výrazy a hodnotí jejich kvalitu. Při vytváření přijímá GFS, souřadnice všech bodů prokládaných dat, parametry vnitřního evolučního algoritmu a meze pro generování konstant.

Výraz z jedince vytvoříme jeho dosazením do metody `ConstructExpression`. Ta výše uvedeným postupem vytvoří příslušný výraz a zároveň si ukládá odkazy na konstanty, byly-li při konstrukci výrazu použity. Všechny mají inicializační hodnotu 0:

```
public IExpression ConstructExpression(int[] individual)
{
    IExpression root = gfs[individual[0]];
    Queue<IExpression> queue = new Queue<IExpression>();
    queue.Enqueue(root);
    // Fronta je prázdná <=> výraz je uzavřen
    for (int i = 1; i < individual.Length && queue.Count != 0; )
    {
        IExpression actual = queue.Dequeue();
        if (actual.UnusedArgumentCount > 0)
        {
            for (int j = 0; j < actual.ArgumentCount; j++, i++)
            {
                IExpression arg = gfs[individual[i]];
```

```
                // Testování konce jedince
                if (root.UnusedArgumentCount + i + arg.UnusedArgumentCount <= individual.Length)
                {
                    actual[j] = arg;
                    queue.Enqueue(arg);
                    if (arg is Constant)
                    {
                        constants.Add((Constant)arg);
                    }
                }
                else
                {
                    // Redukce GFS a nový pokus
                    gfs.Reduce();
                    j--;
                    i--;
                    continue;
                }
            }
        }
    }
    return root;
}
```

Výpis 11: Transformace jedince na výraz

Máme-li vytvořený výraz, přistoupíme k jeho ohodnocení. Pokud výraz neobsahuje konstanty, je vrácen výše definovaný rozdíl mezi prokládanými daty a syntetizovanou funkcí. Pokud však tato funkce obsahuje nekonečna či imaginární části, je nutné vhodnost výrazu příslušně penalizovat:

```
private double GetArea(IExpression expression)
{
    double[] exprValue = expression.GetValue(xs);
    double cv = 1;
    for (int i = 0; i < exprValue.Length; i++)
    {
        double expr = exprValue[i];
        if (Double.IsInfinity(expr) || Double.IsNaN(expr))
        {
            cv *= Penalisation;
            continue;
        }
        cv += Math.Abs(ys[i] - expr);
    }
    return cv - 1;
}
```

Výpis 12: Výpočet rozdílu mezi prokládanými daty a syntetizovanou funkcí

Obsahuje-li však výraz konstanty, musíme jejich hodnoty nejdříve odhadnout. To provedeme pomocí vnořeného evolučního algoritmu. Dimenze problému zde bude počet konstant a hodnoty parametrů jedinců přímo hodnoty konstant:

```
public double ComputeCost(IExpression expression)
{
    int count = constants.Count;
    if (count > 0)
    {
        // Takovýto výraz obsahuje pouze konstanty
        if (count > expression.Size / 2)
            return Double.PositiveInfinity;
        parameters.Dimension = count;
        Specimen specimen = new Specimen(count, innerLow, innerHigh, random);
        // Účelová funkce pro vnitřní evoluční algoritmus
        Func<Individual, double> innerCost = i =>
        {
            double[] ind = (double[])i;
            for (int j = 0; j < count; j++)
            {
                constants[j].Value = ind[j];
            }
            return GetArea(expression);
        };
        EvolutionaryAlgorithm innerEA = null;
        if (parameters is DEParameters)
        {
            innerEA = new DE((DEParameters)parameters, specimen, innerCost, random, null);
        }
        else if (parameters is ESParameters)
        {
            innerEA = new ES((ESParameters)parameters, specimen, innerCost, random, null);
        }
        else if (parameters is SAParameters)
        {
            innerEA = new SA((SAParameters)parameters, specimen, innerCost, random, null);
        }
        else if (parameters is PSOParameters)
        {
            innerEA = new PSO((PSOParameters)parameters, specimen, innerCost, random, null);
        }
        else
        {
         innerEA = new SOMA((SOMAParameters)parameters, specimen, innerCost, random, null);
        }
        Individual best = innerEA.GetBestSolution();
        for (int j = 0; j < count; j++)
        {
            constants[j].Value = best[j];
        }
        return best.Cost;
    }
    return GetArea(expression);
}
```

Výpis 13: Odhad hodnot konstant pomocí vnořeného evolučního algoritmu

4.4 Generátory náhodných čísel

Pro správnou funkci evolučních algoritmů je zapotřebí náhody – náhodných čísel. K jejich generování však nepoužijeme vestavěný generátor pseudonáhodných (dále jen náhodných) čísel, ale Mersenne Twister a generování náhodných čísel pomocí logistické rovnice.

4.4.1 Třída MersenneTwister

`MersenneTwister` je rozšířením třídy `Random`, oproti které poskytuje řadu výhod [13]:

- Extrémně dlouhá perioda
- Lepší vlastnosti rovnoměrného rozložení
- Vysoká efektivita výpočtu čísel

4.4.2 Logistická rovnice

Logistická rovnice představená R. Mayem [14] je formou chaotického systému, který vznikl jako popisný vzorec pro vývoj populace v čase. Na této rovnici lze dokázat, že složité chaotické chování může vznikat i v jednoduché nelineární rovnici. Tuto rovnici popisuje následující vztah:

$$x_{(n+1)} = rx_n(1 - x_n) \tag{4.1}$$

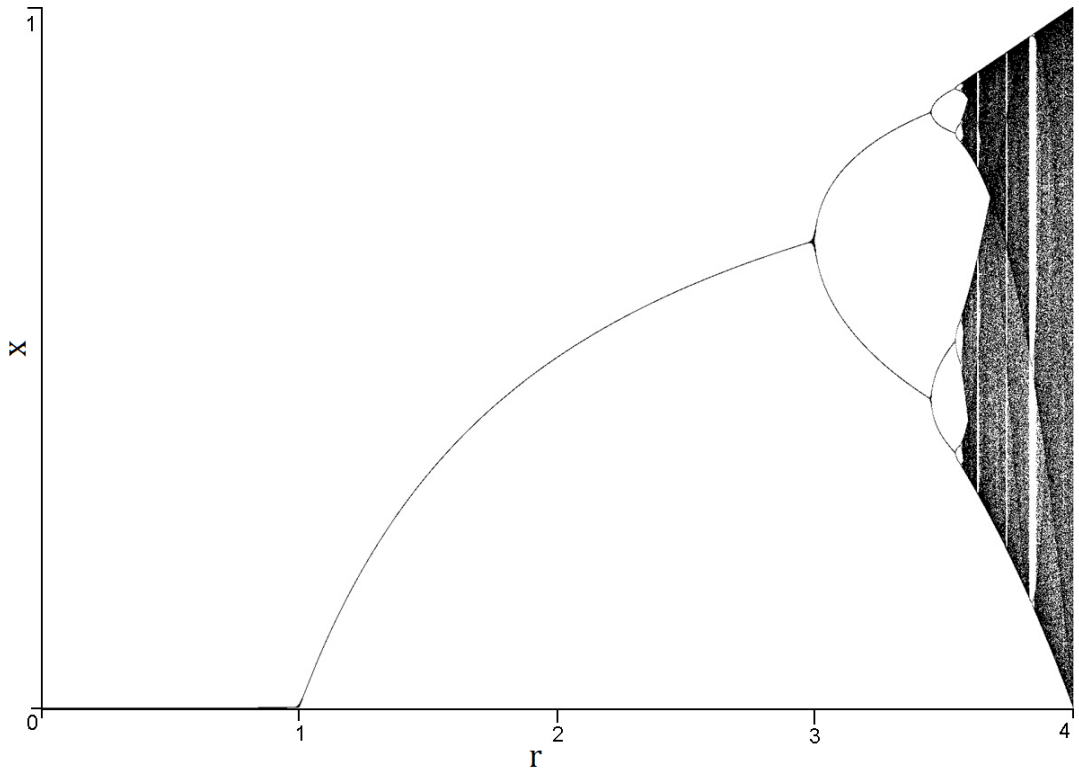
Chování rovnice lze ovlivňovat nastavením velikosti parametru r v rozmezí $\langle 0, 4 \rangle$, kdy při $r = 4$ rovnice vykazuje chaotické chování charakteristické citlivostí na počáteční podmínky.

Tohoto chování lze velmi dobře využít k naprogramování alternativního generátoru náhodných čísel právě pomocí této rovnice. Pro implementaci využijeme, stejně jako u třídy `MersenneTwister`, bázové třídy `Random`, která je součástí .NET Frameworku.

Při testování nově vytvořeného generátoru náhodných čísel však narazíme na jeden nedostatek – zaokrouhlování. Datové typy `Single` a `Double` v C# (a také v ostatních jazycích) mají omezenou přesnost výpočtu.

Vyjde-li nám jako výsledek logistické rovnice číslo 0,50000001, bude výsledkem rovnice v dalším kroku číslo 1, nikoliv 0,9999999999999996, a posloupnost se nám zastaví. Tato skutečnost je dána tím, že čísla jsou v paměti uložená s omezenou přesností a při jejím překročení jsou jednoduše zaokrouhlena.

Řešením tohoto problému by mohlo být použití datového typu s větší přesností (`Decimal`). Nevýhodou tohoto řešení je ale citelné zpomalení chodu programu. Další možností je snížení parametru r např. na 3,999999 nebo použití počáteční hodnoty např. $x = 0,02$, u které se tento problém při experimentech neobjevil.



Obrázek 4.12: Bifurkační diagram logictické rovnice (převzato z [15])

4.4.3 Třída RandomExtensions

Jazyk C# od verze 3.0 (.NET Framework 3.5) přináší novou technologii tzv. Extension metod. V podstatě se jedná o způsob, jak zvenčí přidat třídě instanční metodu, aniž bychom tuto třídu museli modifikovat. Tuto vlastnost můžeme využít při programování algoritmů ES nebo SA, které generují množiny sousedů pomocí Gaussova mutačního operátoru – nepoužívají tedy klasické rovnoměrné rozdělení. Jelikož třída `MersenneTwister` i `LogisticMap` dědí ze třídy `Random` a používají tak stejné metody, můžeme vytvořit Extension metodu rozšiřující právě třídu `Random`. Klíčem k jejímu vytvoření je umístění této statické metody do statické třídy a použití slova `this`:

```
public static class RandomExtensions
{
    public static double NextDoubleGaussian(this Random random,
                    double mean, double variance)
    {
        double u1 = random.NextDouble();
        double u2 = random.NextDouble();

        double stdNormal = Math.Sqrt(−2.0 ∗ Math.Log(u1)) ∗ Math.Sin(2.0 ∗ Math.PI ∗ u2);
        double normal = mean + variance ∗ stdNormal;

        return normal;
    }
}
```

Výpis 14: Extension metoda třídy Random

4.5 Ostatní třídy

4.5.1 Třída TestFunctions

Každý námi naprogramovaný evoluční algoritmus je třeba otestovat na množině testovacích funkcí. Tak poznáme, naprogramovali jsme jej správně, a zda je schopen optimalizovat složitější problémy. Mezi implementované testovací funkce patří:

- 1., 2., 3. a 4. De Jongova funkce
- Schwefelova funkce
- Rastriginova funkce
- Griewangkova funkce
- Michalewiczova funkce
- Plato vajec (Egg Holder)
- Ackleyho funkce I a II
- Sinová obálková sinusoidální funkce

4.5.2 Třída FunctionGenerator

Tato třída obsahuje jednu metodu pro generování aproximovaných dat pro analytické programování. Jako argumenty funkce dosadíme interval generování bodů, počet generovaných bodů, generovanou funkci a název souboru, do kterého budou hodnoty uloženy:

```
FunctionGenerator.Generate(−1, 1, 50,
                    x => Math.Pow(x, 5) − 2 ∗ Math.Pow(x, 3) + x, "quintic.txt");
FunctionGenerator.Generate(−1, 1, 50,
                    x => Math.Pow(x, 6) − 2 ∗ Math.Pow(x, 4) + x ∗ x, "sextic.txt");
```

Výpis 15: Ukázka generování dat pro AP

4.5.3 Třída EAParametersParser

Třída obsahuje metody, které z konfiguračního XML souboru extrahují potřebné parametry pro naprogramované evoluční algoritmy.

4.5.4 Třída FunctionParser

V konfiguračním XML souboru jsou kromě parametrů evolučních algoritmů definovány také funkce, které budou v AP v GFS použity. Jedná se o základní již naprogramované funkce, stačí zde proto uvést pouze jejich název.

Pokud bychom ovšem chtěli do GFS přidat předem nenaprogramovanou funkci, můžeme ji definovat přímo v konfiguračním souboru. Stačí zde zadat název funkce, počet argumentů a její předpis přímo v jazyce C#. Funkce je následně za běhu zkompilována a použita stejně, jako by byla přímo naprogramována.

4.5.5 Třída ChartForm

Třída ChartForm slouží k zobrazování grafů. Můžeme si tak velice jednoduše zobrazit výslednou syntetizovanou funkci v porovnání s aproximovanými daty. Můžeme zde přidat jak prostá data, tak celého jedince a z jeho výrazu nechat vygenerovat graf. Tento graf můžeme zobrazit na obrazovce nebo jej uložit do souboru.

5 Experimenty

Nyní, když máme vše neprogramováno, přistoupíme k testování algoritmu analytického programování na dvou vybraných problémech, přičemž bude spuštěno na všech pěti evolučních algoritmech, které jsme si popsali.

Jak již bylo zmíněno výše, v této práci byla implementována verze AP_{meta}. Jako vnitřní evoluční algoritmus můžeme dosadit opět jeden z námi implementovaných algoritmů. Dostáváme tak 25 kombinací evolučních algoritmů, jejichž kvalitu budeme porovnávat.

Dalším kritériem porovnávání bude použití dvou různých generátorů náhodných čísel. Každou takovouto kombinaci (pět vnějších EA $\times$ pět vnitřních EA $\times$ dva generátory náhodných čísel $\times$ dva vybrané problémy) provedeme 50$\times$. Celkem tedy provedeme 5 000 simulací.

Tyto simulace budou provedeny na dvou klasických desktopových PC s čtyřjádrovým procesorem Intel Core i7-3770 o frekvenci 3,4 GHz (3,9 GHz Turbo) s Hyper-Threadingem (HT). Jelikož je celý projekt naprogramovaný jako jednovláknová aplikace, můžeme bez problému spustit např. 14 jeho instancí s různými parametry najednou.

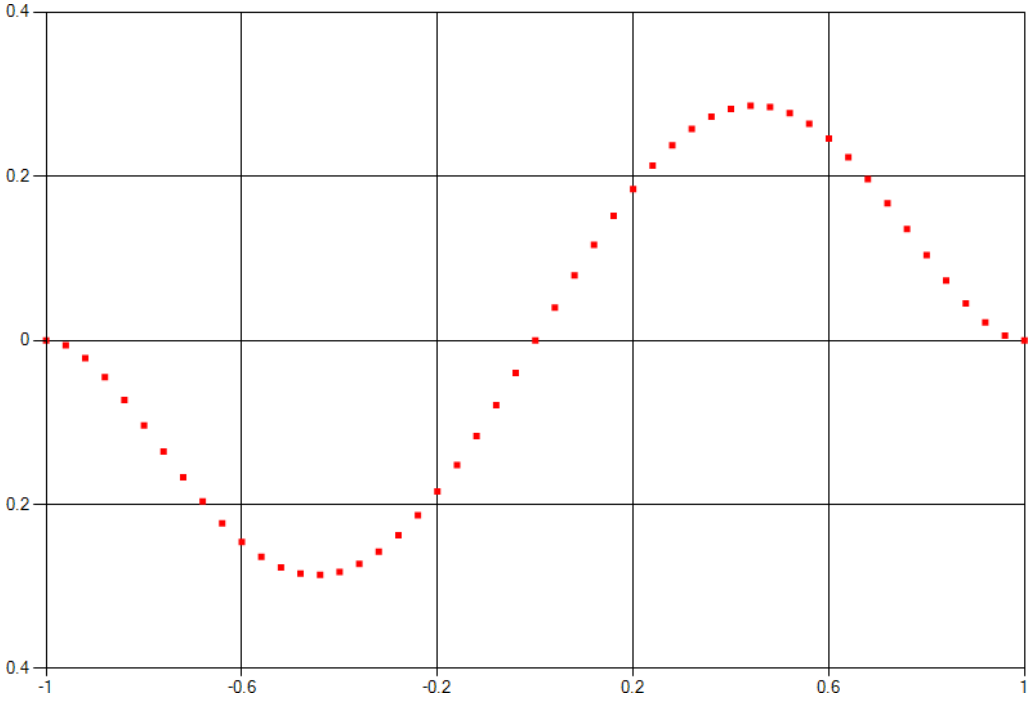
5.1 Vybrané testovací problémy

Jako testovací problémy byly zvoleny dva [1]: Quintic a Sextic. Quintic problém je dán vztahem 5.1 a problém Sextic vztahem 5.2.

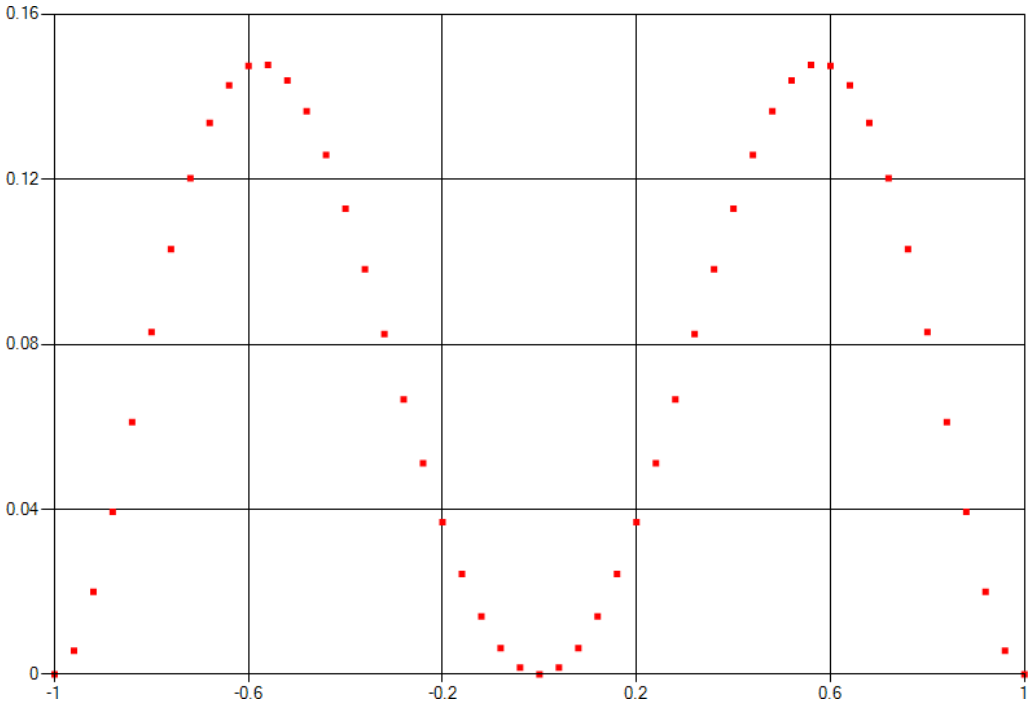
$$x^5 - 2x^3 + x \tag{5.1}$$

$$x^6 - 2x^4 + x^2 \tag{5.2}$$

Z každého z těchto dvou vztahů bude vygenerováno 50 bodů v intervalu $\langle -1, 1 \rangle$, které budou použity jako aproximovaná data v AP.



Obrázek 5.1: Vygenerované body z Quintic problému



Obrázek 5.2: Vygenerované body ze Sextic problému

5.2 Nastavení parametrů EA a AP

Nastavení parametrů evolučních algoritmů bylo inspirováno [1]. Všechny evoluční algoritmy budou mít stejnou dimenzi problému a také budou mít nastaven stejný počet jedinců v populaci.

Pro porovnání výsledků evolučních algoritmů je velice důležité nastavit jejich parametry tak, aby měly ve výsledku stejný počet ohodnocení účelové funkce (Cost Function Evaluations, CFE). V této práci byl pro vnější evoluci zvolen CFE = 30 000 a pro vnitřní evoluci byl zvolen CFE = 5 000.

5.2.1 Parametry ES

Parametry ES pro vnější, resp. vnitřní evoluci byly nastaveny takto:

Parametr	**Hodnota (vnější EA)**	**Hodnota (vnitřní EA)**
Dimension	50	-
Parents	100	50
Offspring	10	5
Iterations	30	20
Deviation	0,723	0,723
Strategy	,	,
Recombination Parents	4	4

Tabulka 5.1: Parametry ES

5.2.2 Parametry SA

Parametry SA pro vnější, resp. vnitřní evoluci byly nastaveny takto:

Parametr	**Hodnota (vnější EA)**	**Hodnota (vnitřní EA)**
Dimension	50	-
Population	100	50
Neighbors	10	5
Deviation	0,723	0,723
TStart	5	1
TFinal	0,01	0,011
Decr	0,8	0,8
Repetitions	30	30

Tabulka 5.2: Parametry SA

5.2.3 Parametry PSO

Parametry PSO pro vnější, resp. vnitřní evoluci byly nastaveny takto:

Parametr	Hodnota (vnější EA)	Hodnota (vnitřní EA)
Dimension	50	-
Population	100	50
Migrations	300	100
c1	2	2
c2	2	2
VMax	0,3	0,3
wStart	0,8	0,4
wEnd	0,8	0,4

Tabulka 5.3: Parametry PSO

5.2.4 Parametry DE

Parametry DE pro vnější, resp. vnitřní evoluci byly nastaveny takto:

Parametr	Hodnota (vnější EA)	Hodnota (vnitřní EA)
Dimension	50	-
Population	100	50
Generations	300	100
CR	0,7	0,7
F	0,85	0,85

Tabulka 5.4: Parametry DE

5.2.5 Parametry SOMA

Parametry SOMA pro vnější, resp. vnitřní evoluci byly nastaveny takto:

Parametr	Hodnota (vnější EA)	Hodnota (vnitřní EA)
Dimension	50	-
Population	100	50
Migrations	12	11
PathLength	3	2
Step	0,11	0,21
PRT	0,1	0,1
MinDiv	-0,001	-0,001
Strategy	AllToOne	AllToOne

Tabulka 5.5: Parametry SOMA

5.2.6 Parametry AP

K syntéze byla použita $GFS = \{+, -, *, /, K\}$, kde x je nezávisle proměnná a K reprezentuje obecnou konstantu. Práh posíleného hledání byl nastaven pro Quintic problém na hodnotu 2,5 a pro Sextic problém na hodnotu 1,5.

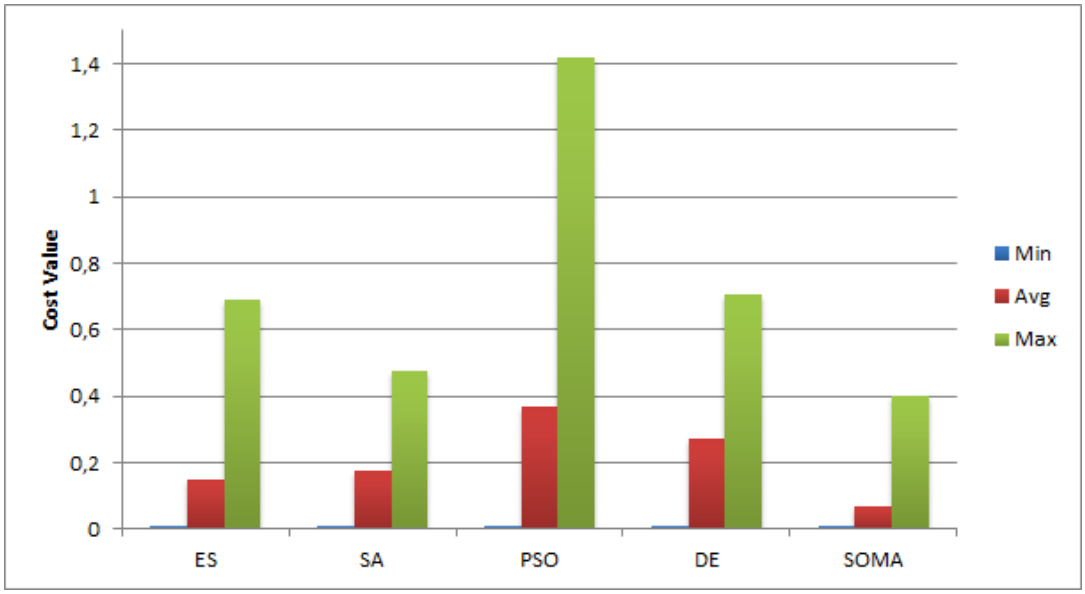
5.3 Výsledky testování

5.3.1 Porovnání vhodností nejlepších jedinců

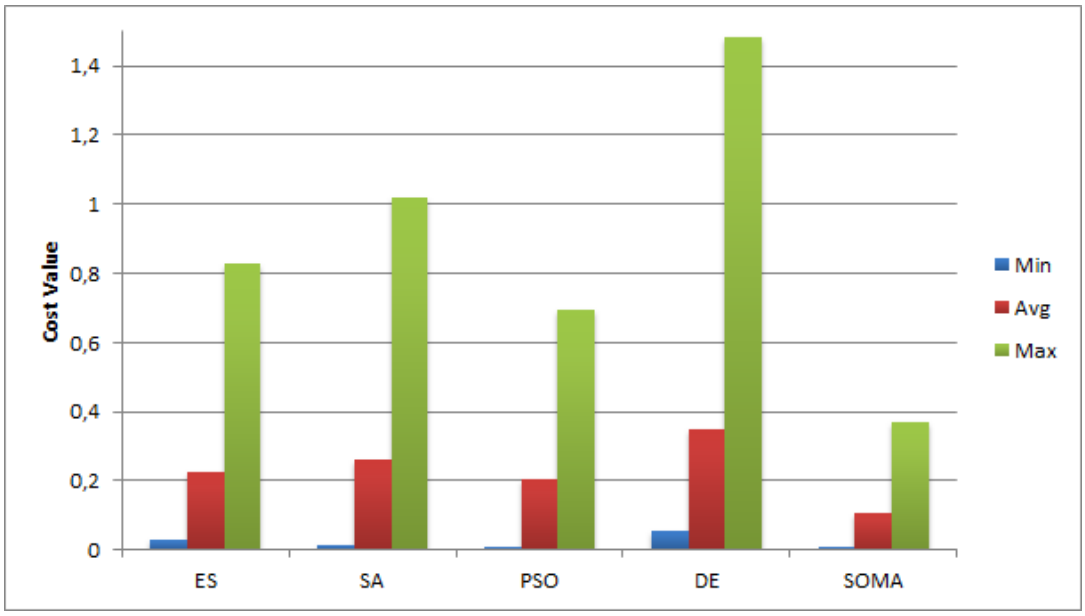
Testování analytického programování a evolučních algoritmů proběhlo přesně podle podmínek definovaných výše. V následující tabulce a grafech si nyní ukážeme porovnání výsledných vhodností nejlepších jedinců pro dva problémy a dva generátory náhodných čísel:

Algoritmus	Problém	Generátor	Min	Avg	Max
ES	Quintic	Twister	$9,887619 \cdot 10^{-4}$	$1,490182 \cdot 10^{-1}$	$6,910020 \cdot 10^{-1}$
SA	Quintic	Twister	$8,157133 \cdot 10^{-4}$	$1,730835 \cdot 10^{-1}$	$4,755063 \cdot 10^{-1}$
PSO	Quintic	Twister	$4,440892 \cdot 10^{-16}$	$3,660675 \cdot 10^{-1}$	$1,417831 \cdot 10^{0}$
DE	Quintic	Twister	$6,210946 \cdot 10^{-3}$	$2,715824 \cdot 10^{-1}$	$7,064366 \cdot 10^{-1}$
SOMA	Quintic	Twister	$6,661338 \cdot 10^{-16}$	$6,979811 \cdot 10^{-2}$	$3,999587 \cdot 10^{-1}$
ES	Quintic	Logistic	$2,811095 \cdot 10^{-2}$	$2,260901 \cdot 10^{-1}$	$8,267810 \cdot 10^{-1}$
SA	Quintic	Logistic	$1,338242 \cdot 10^{-2}$	$2,610050 \cdot 10^{-1}$	$1,018202 \cdot 10^{0}$
PSO	Quintic	Logistic	$6,661338 \cdot 10^{-16}$	$2,029268 \cdot 10^{-1}$	$6,925307 \cdot 10^{-1}$
DE	Quintic	Logistic	$5,621102 \cdot 10^{-2}$	$3,517544 \cdot 10^{-1}$	$1,482279 \cdot 10^{0}$
SOMA	Quintic	Logistic	$1,008551 \cdot 10^{-3}$	$1,056129 \cdot 10^{-1}$	$3,702667 \cdot 10^{-1}$
ES	Sextic	Twister	$3,703753 \cdot 10^{-3}$	$1,337584 \cdot 10^{-1}$	$4,028901 \cdot 10^{-1}$
SA	Sextic	Twister	$4,440892 \cdot 10^{-16}$	$1,668817 \cdot 10^{-1}$	$6,068369 \cdot 10^{-1}$
PSO	Sextic	Twister	$6,038816 \cdot 10^{-4}$	$2,713840 \cdot 10^{-1}$	$1,094095 \cdot 10^{0}$
DE	Sextic	Twister	$5,626506 \cdot 10^{-3}$	$2,325586 \cdot 10^{-1}$	$8,527469 \cdot 10^{-1}$
SOMA	Sextic	Twister	$1,853924 \cdot 10^{-3}$	$6,211643 \cdot 10^{-2}$	$3,159944 \cdot 10^{-1}$
ES	Sextic	Logistic	$1,667438 \cdot 10^{-2}$	$2,222940 \cdot 10^{-1}$	$5,625207 \cdot 10^{-1}$
SA	Sextic	Logistic	$6,282266 \cdot 10^{-2}$	$2,523713 \cdot 10^{-1}$	$8,019849 \cdot 10^{-1}$
PSO	Sextic	Logistic	$3,510573 \cdot 10^{-4}$	$1,875989 \cdot 10^{-1}$	$7,368225 \cdot 10^{-1}$
DE	Sextic	Logistic	$7,832922 \cdot 10^{-2}$	$3,315250 \cdot 10^{-1}$	$8,610967 \cdot 10^{-1}$
SOMA	Sextic	Logistic	$6,803753 \cdot 10^{-3}$	$1,141296 \cdot 10^{-1}$	$3,575043 \cdot 10^{-1}$

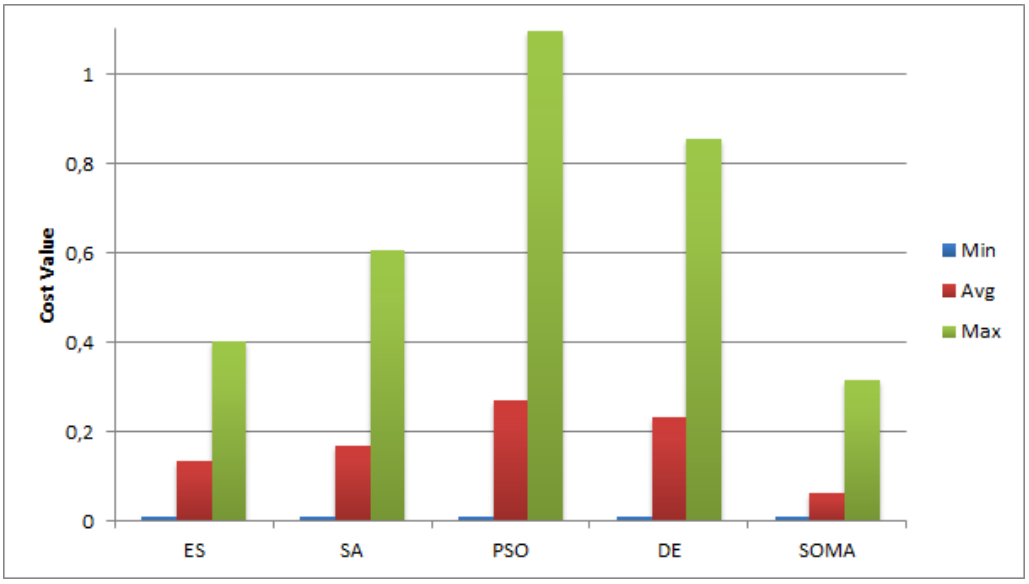
Tabulka 5.6: Porovnání výsledných vhodností nejlepších jedinců



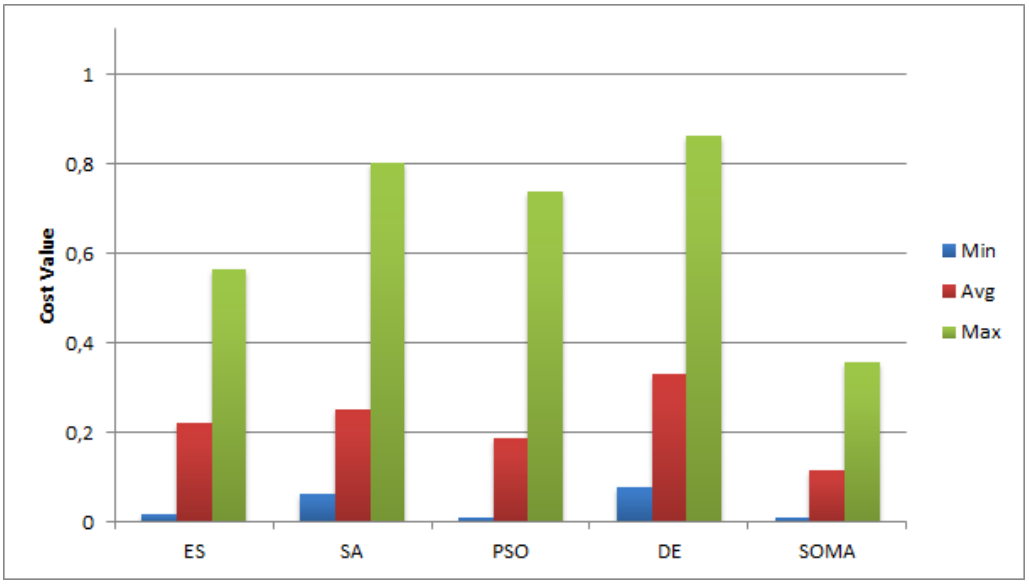
Obrázek 5.3: Vhodnosti jedinců pro Quintic problém a generátor MersenneTwister



Obrázek 5.4: Vhodnosti jedinců pro Quintic problém a generátor LogisticMap



Obrázek 5.5: Vhodnosti jedinců pro Sextic problém a generátor MersenneTwister

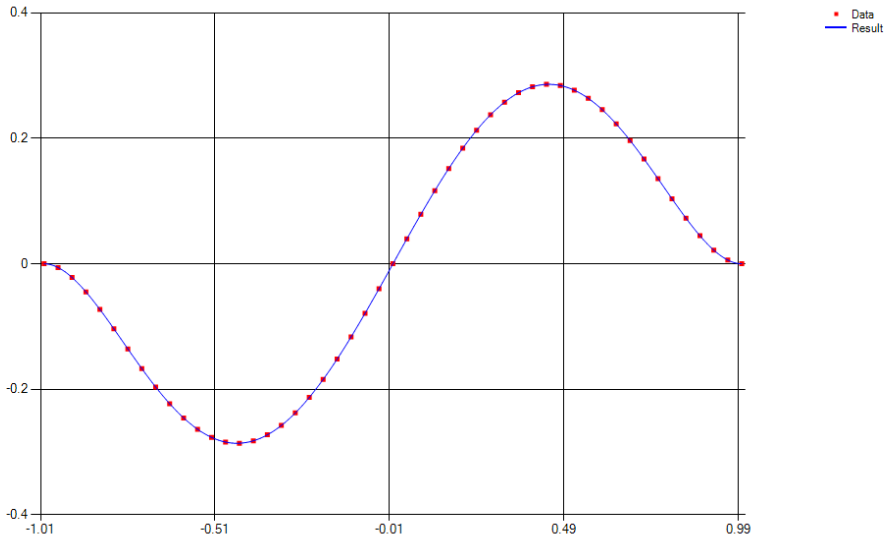


Obrázek 5.6: Vhodnosti jedinců pro Sextic problém a generátor LogisticMap

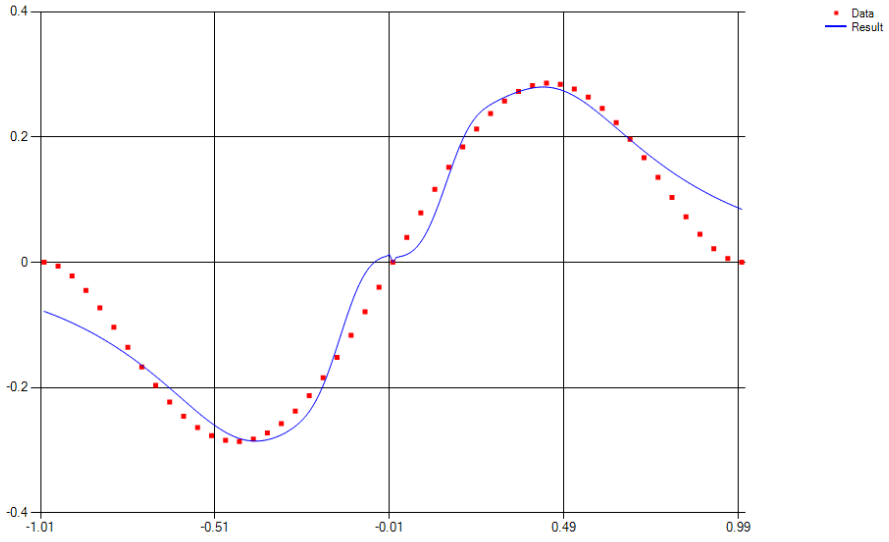
V těchto srovnáních jsme nerozlišovali algoritmy vnitřní evoluce – výsledky jsou pro každý algoritmus vnější evoluce sjednoceny. Podrobnější grafy výsledků testů jsou umístěny v příloze. V příloze jsou také uvedeny grafy průběhů všech kombinací evolučních algoritmů. U grafů některých průběhů algoritmu PSO bylo záměrně upraveno měřítko z důvodu občasného výskytu velmi vysokých hodnot účelové funkce.

5.3.2 Vybrané grafy funkcí výsledných jedinců

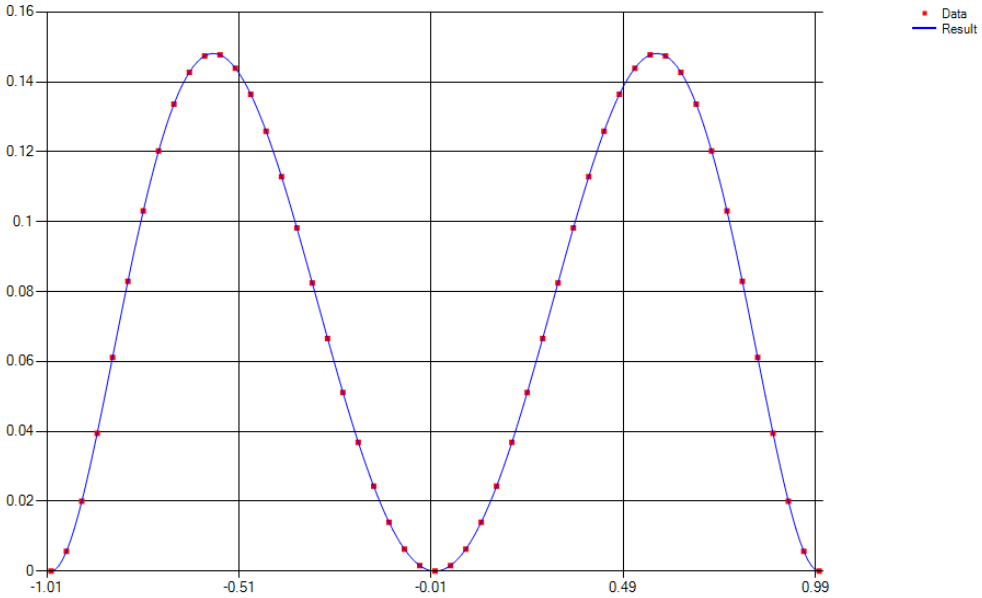
Pro zajímavost si můžeme ukázat grafy funkcí výsledných (nejlepších a nejhorších) jedinců pro oba dva problémy:



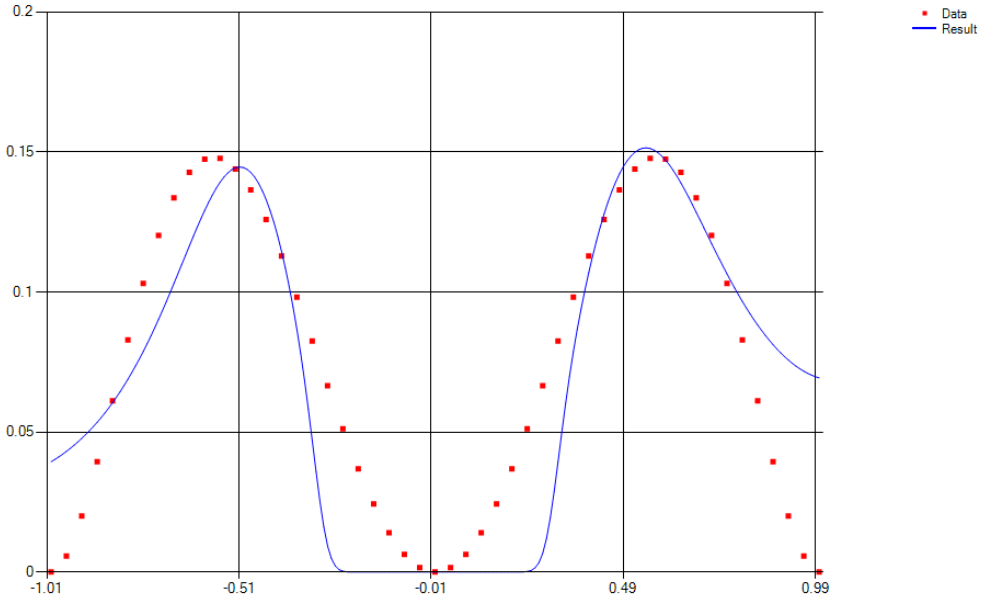
Obrázek 5.7: Nejlepší jedinec ze všech simulací pro Quintic problém



Obrázek 5.8: Nejhorší jedinec ze všech simulací pro Quintic problém



Obrázek 5.9: Nejlepší jedinec ze všech simulací pro Sextic problém



Obrázek 5.10: Nejhorší jedinec ze všech simulací pro Sextic problém

6 Závěr

Cílem této práce bylo vytvoření algoritmu analytického programování, jedné z metod symbolické regrese, s čímž souvisí i problematika evolučních algoritmů, pod kterými AP pracuje.

V první kapitole jsme se seznámili s evolučními algoritmy, speciální třídou optimalizačních algoritmů, které jsou inspirovány biologickými procesy. V rámci této kapitoly jsme si uvedli pět evolučních algoritmů, které jsou v práci implementovány: evoluční strategie, simulované žíhání, rojení částic, diferenciální evoluce a samoorganizující se migrační algoritmus.

Ve druhé kapitole jsme se seznámili s problémem symbolické regrese a její aplikace. Nastínili jsme si tři algoritmy: genetické programování, gramatickou evoluci a analytické programování, které bylo popsáno podrobněji. Popsali jsme si zde princip činnosti AP, souvislost s evolučními algoritmy a techniku posíleného hledání. Předvedli jsme si také tři verze AP, které se liší v principu generování a odhadování konstant, a pro implementaci jsme vybrali verzi AP_{meta}, která odhaduje konstanty pomocí vnitřního evolučního algoritmu.

Třetí kapitola pojednává o samotné implementaci evolučních algoritmů, analytického programování a reprezentaci výrazů. Pozornost zde byla věnována také výběru generátoru náhodných čísel. Nebyla zde použita standardní třída .NET Frameworku, ale třída MersenneTwister a dále generátor náhodných čísel pomocí logistické rovnice. U obou generátorů jsme pak zkoumali jejich vliv na výkon evolučních algoritmů.

Při programování celého projektu byla snaha implementovat jej tak, aby byl snadno rozšířitelný o další evoluční algoritmy a výrazy. Algoritmus AP je prozatím implementován pouze pro použití k aproximaci dat vhodnou matematickou formulí. Naprogramovat verzi AP řešící jiný problém a začlenit ji do projektu by však mělo být velmi snadné.

Během implementace posíleného hledání u AP se objevil problém s narůstáním délky výrazů, kdy stromy reprezentující tyto výrazy měly mnohdy i statisíce uzlů. Z tohoto důvodu nejsou jedinci porovnáváni pouze pomocí hodnoty účelové funkce, ale i pomocí délky výrazu (např. výraz $\cos(x)$ označíme lepší než výraz $\cos(x + x - x)$). Tento přístup pomohl v mnoha případech délku výsledných jedinců omezit. Zde ale vyvstává otázka, zda je tento přístup skutečně lepší a zda by se v některých případech tito „horší“ jedinci ukázali jako lepší základ pro nastávající generace jedinců.

Při nastavování parametrů evolučních algoritmů byla snaha o to, aby tyto algoritmy pak bylo možné objektivně porovnat. To zajistíme pouze nastavením parametrů tak, aby měly ve výsledku stejný počet ohodnocení účelové funkce (CFE). V této práci byl pro vnější evoluci zvolen CFE = 30 000 a pro vnitřní evoluci byl zvolen CFE = 5 000. Jak se po skončení simulací ukázalo, u algoritmu simulovaného žíhání nebyly parametry odhadnuty správně a bylo provedeno někdy i 45 000 CFE (což ale vzhledem k výsledkům simulací uvedeným níže nevadí).

Ze sjednocených grafů výsledků 5.3, 5.4, 5.5 a 5.6 lze usoudit, že absolutně nejlepším námi implementovaným algoritmem řešícím problémy Quintic a Sextic je algoritmus SOMA. Na druhém místě se umístil algoritmus ES, dále pak SA, DE a poslední PSO.

V případě použití generátoru náhodných čísel pomocí logistické rovnice se výrazně zlepšily výsledky algoritmu PSO a pořadí bylo následující: SOMA, PSO, ES, SA, DE.

Porovnáme-li generátor náhodných čísel Mersenne Twister a generátor náhodných čísel pomocí logistické rovnice, byl ve většině případů (ES, SA, DE, SOMA) pro oba dva problémy lepší generátor Mersenne Twister. Výjimkou je zde algoritmus PSO, pro který bylo pro oba dva problémy mnohem výhodnější použít generátoru náhodných čísel pomocí logistické rovnice.

Z grafů porovnání výsledků vnitřních evolučních algoritmů (C) můžeme usoudit, že nejlepším algoritmem vnitřní evoluce pro odhadování konstant je algoritmus PSO (což je plně v souladu s No Free Lunch teorémem).

Grafy průběhů evolučních algoritmů (D) zachycují průběhy všech kombinací evolučních algoritmů pro oba problémy a oba generátory náhodných čísel. Z těchto grafů můžeme také vypozorovat přibližný počet ohodnocení účelové funkce k dosažení výsledků, který se pohybuje v rozmezí 5 000 – 15 000.

U algoritmu DE lze vidět rychlejší konvergenci k suboptimálnímu řešení (asi po 3 000 CFE), kdy další vývoj populace nepokračuje, přestože populace neztratila diverzitu. Výsledky přitom nejsou tak kvalitní jako u ostatních algoritmů.

Celý tento projekt byl vytvořen jako jednovláknová aplikace. Jeden běh simulace každého algoritmu trval přes půl hodiny. Velmi žádoucí by tedy byla jeho paralelizace např. s využitím technologie NVIDIA CUDA. Dalším možným vylepšením projektu by mohla být také implementace analytického programování pro jiné účely, než je aproximace dat matematickou funkcí.

7 Reference

[1] ZELINKA, Ivan, Zuzana OPLATKOVÁ, Miloš ŠEDA, Pavel OŠMERA a František VČELAŘ. *Evoluční výpočetní techniky: principy a aplikace*. 1. české vyd. Praha: BEN, 2009, 534 s. ISBN 978-80-7300-218-3.

[2] POSPÍCHAL, Jiří, Vladimír KVASNIČKA a Peter TIŇO. *Evolučné algoritmy*. 1. vyd. Bratislava: Slovenská technická univerzita, 2000, 215 s. Edícia vysokoškolských učebníc. ISBN 80-227-1377-5.

[3] KIRKPATRICK, Scott, Charles D. GELATT a Mario P. VECCHI. *Optimization by Simulated Annealing* [online]. [cit. 2014-04-22]. Dostupné z: `http://leonidzhukov.net/hse/2013/stochmod/papers/KirkpatrickGelattVecchi83.pdf`

[4] KENNEDY, James, Russell C EBERHART a Yuhui SHI. *Swarm intelligence*. San Francisco: Morgan Kaufmann, 2001, xxvii, 512 s. ISBN 15-586-0595-9.

[5] STORN, Rainer a Kenneth PRICE. *Differential Evolution – a simple and efficient heuristic for global optimization over continuous spaces* [online]. 1997 [cit. 2014-04-22]. Dostupné z: `http://www1.icsi.berkeley.edu/~storn/TR-95-012.pdf`

[6] ZELINKA, Ivan. *SOMA – Self Organizing Migrating Algorithm: New optimization techniques in engineering*. Berlin: Springer, 2004, xxii, 712 s. ISBN 3-540-20167X.

[7] KOZA, John R. *Genetic programming II: automatic discovery of reusable programs*. Cambridge: MIT Press, c1994, xx, 746 s. Complex adaptive systems. ISBN 02-621-1189-6.

[8] O'NEILL, Michael a Conor RYAN. *Grammatical evolution: evolutionary automatic programming in an arbitrary language*. Boston: Kluwer, c2003, xvi, 144 s. Complex adaptive systems. ISBN 14-020-7444-1.

[9] JANČAR, Petr. *Teoretická informatika* [online]. Ostrava: Vysoká škola báňská – Technická univerzita, 2007, 1 CD-R [cit. 2014-04-22]. ISBN 978-80-248-1487-2. Dostupné z: `http://www.cs.vsb.cz/jancar/TEORET-INF/ti-text.2010-01-20.pdf`

[10] ZELINKA, Ivan. *Analytic Programming by Means of Soma Algorithm*. ICICIS'02, First International Conference of Intelligent Computing and Information Systems, Egypt, Cairo, 2002

[11] *Nonlinear Least Squares Fitting*. WOLFRAM RESEARCH, Inc. Wolfram MathWorld [online]. [cit. 2014-04-22]. Dostupné z: `http://mathworld.wolfram.com/NonlinearLeastSquaresFitting.html`

[12] *Better random numbers in .Net: an updated Mersenne Twister in C#*. MICROSOFT CORPORATION. MSDN [online]. 2008 [cit. 2014-04-22]. Dostupné z: `http://archive.msdn.microsoft.com/MersenneTwister`

[13] ALBRECHT, Josh. *A Comparison of Mersenne Twister and Linear Congruential Random Number Generators*. [online]. [cit. 2014-04-22]. Dostupné z: http://people.cs.pitt.edu/~jsa8/math_project.pdf

[14] MAY, Robert M. *Stability and complexity in model ecosystems*. 1st Princeton landmarks in biology ed. Princeton: Princeton University Press, 2001, xxxiv, 265 p. ISBN 06-910-8861-6.

[15] WHELTON, Dean. *Characteristic polynomials traversing the bifurcation diagram of the logistic map*. [online]. [cit. 2014-04-22]. Dostupné z: http://welbog.homeip.net/~inferno/math/pmath370/

[16] *Extension Methods: C# Programming Guide*. MICROSOFT CORPORATION. MSDN [online]. [cit. 2014-04-22]. Dostupné z: http://msdn.microsoft.com/en-us//library/bb383977.aspx

A Obsah DVD

Následující tabulka popisuje umístění souborů na DVD a jejich popis.

Adresář	**Popis**
AnalyticalProgramming	Zdrojové kódy analytického programování
Documentation	HTML dokumentace projektu
Analytic Programming.txt	Text diplomové práce

Tabulka A.1: Obsah DVD

B Uživatelská příručka

B.1 Aplikace

Samotná aplikace a všechny potřebné soubory se nachází ve složce projektu:

```
AnalyticalProgramming/bin/Debug
```

Tato složka obsahuje následující důležité soubory:

- AnalyticalProgramming.exe – hlavní aplikace
- config.xml – konfigurační soubor aplikace
- quintic.txt, sextic.txt apod. – zdrojová data

B.2 Konfigurace

B.2.1 Parametry evolučních algoritmů

```
<outerEA name="DE" simulations="50" />

<innerEA name="DE" constLow="-15" constHigh="15" />
```

Nastavením parametru `simulations` určíme počet opakování vnější evoluce (počet simulací). Nastavením parametru `name` zvolíme evoluční algoritmus pro vnější, popř. vnitřní evoluci. Hodnoty parametru `name` mohou být následující:

- **DE** – Differential Evolution
- **ES** – Evolution Strategies
- **SA** – Simulated Annealing
- **PSO** – Particle Swarm Optimization
- **SOMA** – Self Organising Migrating Algorithm

Dále musíme nastavit parametry příslušného evolučního algoritmu:

- **DE**
  - **dim** – dimenze problému
  - **pop** – velikost populace
  - **gen** – počet generací
  - **cr** – práh křížení
  - **f** – mutační konstanta

- **ES**
  - **dim** – dimenze problému
  - **parents** – počet rodičů
  - **offspring** – počet potomků každého rodiče
  - **iter** – počet iterací
  - **deviation** – směrodatná odchylka pro Gaussovo normální rozdělení
  - **strategy** – strategie výběru jedinců do nové populace
    - **+** – výběr z množiny rodičů a potomků
    - **,** – výběr pouze z množiny potomků
  - **recombinationParents** – počet rodičů pro rekombinaci (pokud je menší než 2, k rekombinaci nedochází)
- **SA**
  - **dim** – dimenze problému
  - **pop** – velikost populace
  - **neighbors** – počet sousedů aktuálního řešení
  - **deviation** – směrodatná odchylka pro Gaussovo normální rozdělení
  - **tStart** – počáteční teplota
  - **tFinal** – konečná teplota
  - **decr** – redukční faktor teploty
  - **repetitions** – počet opakování Metropolisova algoritmu
- **PSO**
  - **dim** – dimenze problému
  - **pop** – velikost populace
  - **mig** – počet migrací
  - **c1**, **c2** – učící faktory
  - **vMax** – maximální rychlost jedince
  - **wStart** – počáteční hodnota setrvačnosti
  - **wEnd** – koncová hodnota setrvačnosti
- **SOMA**
  - **dim** – dimenze problému
  - **pop** – velikost populace
  - **mig** – počet migrací
  - **pathL** – délka cesty jednice

  - **step** – délka kroku jedince
  - **prt** – perturbace
  - **minDiv** – minimální diverzita
  - **strategy** – strategie algoritmu
    - **ao** – AllToOne
    - **aa** – AllToAll
    - **aaa** – AllToAllAdaptive

U vnitřní evoluce přibyly další dva parametry:

- **constLow** – spodní hranice pro generování konstant
- **constHigh** – horní hranice pro generování konstant

Dimenzi zde také v podstatě nemá smysl nastavovat (proto hodnota -1). Dimenze problému se vždy nastaví za běhu podle aktuálního množství konstant k evoluci.

B.2.2 Množina funkcí

Pod uzlem `<gfs>` máme možnost nastavit, jaké funkce budou v AP použity. Na výběr máme (prozatím) z následujících funkcí:

- `<f>+</f>` – sčítání ($x + y$)
- `<f>-</f>` – odčítání ($x - -y$)
- `<f>*</f>` – násobení ($x * y$)
- `<f>/</f>` – dělení (x/y)
- `<f>pow</f>` – mocnina (x^y)
- `<f>sin</f>` – sinus ($\sin(x)$)
- `<f>cos</f>` – cosinus ($\cos(x)$)
- `<f>tan</f>` – tangens ($\tan(x)$)
- `<f>x</f>` – nezávisle proměnná x
- `<f>c</f>` – konstanta K

Pokud danou funkci v GFS nechceme, můžeme ji odstranit nebo zakomentovat. Pokud nám předdefinované funkce nestačí, můžeme si nadefinovat své vlastní funkce v tomto formátu (stačí nám k tomu elementární znalost jazyka C#):

```
<f argCount="1" name="exp">Math.Exp(x)</f>

<f argCount="2" name="log">Math.Log(x, y)</f>
```

Parametrem `argCount` určíme počet parametrů funkce. Parametr `name` představuje název funkce. Uvnitř uzlu definujeme funkci.

B.2.3 I/O

```
<data fileName="x" />
```

Nastavením parametru `fileName` u uzlu `data` zvolíme soubor s daty, která budou použita v AP. Data budou reálná čísla. Formát dat bude následující:

x_1

$f(x_1)$

x_2

$f(x_2)$

...

x_n

$f(x_n)$

Výsledky simulací se automaticky ukládají do složky output v následujícím formátu názvu souboru:

```
O_I_P_R_S_T
```

- **O** – název vnějšího evolučního algoritmu
- **I** – název vnitřního evolučního algoritmu
- **P** – název problému
- **R** – druh generátoru náhodných čísel
- **S** – pořadí simulace
- **T** – typ souboru
  - **process** – průběh simulace (iterace, cost value nejlepšího a nejhoršího jedince, počet ohodnocení účelových funkcí)
  - **output** – výstup simulace (parametry evolucí, počet ohodnocení účelové funkce, nejlepší cost value, výsledný výraz)
  - **graph** – graf výsledného výrazu

B.2.4 Další nastavení

```
<random generator="x" />
```

Nastavením parametru `generator` uzlu `random` zvolíme generátor náhodných čísel, který bude v programu použit. Máme na výběr z těchto možností:

- **default** – bude použit výchozí generátor náhodných čísel z .NET Frameworku
- **twister** – bude použit generátor Mersenne Twister
- **logistic** – bude použit generátor náhodných čísel pomocí logistické rovnice

```
<ea terminationCost="x" />
```

Nastavením parametru `terminationCost` uzlu `ea` zvolíme hodnotu Cost funkce, při které zastavíme evoluci. Tento parametr vyřadíme nastavením záporné hodnoty.

```
<reinforcedSearch threshold="x" />
```

Nastavením parametru `threshold` uzlu `reinforcedSearch` zvolíme počáteční práh, po jehož překročení přidáme aktuálně syntetizovaný program do GFS. Nastavením záporné hodnoty posílené prohledávání vypneme.

C Porovnání výsledků vnitřních evolučních algoritmů

Algoritmus	Problém	Generátor	Min	Avg	Max
ES	Quintic	Twister	$4,922437 \cdot 10^{-3}$	$1,433374 \cdot 10^{-1}$	$3,342170 \cdot 10^{-1}$
SA	Quintic	Twister	$1,945456 \cdot 10^{-2}$	$1,727474 \cdot 10^{-1}$	$6,486075 \cdot 10^{-1}$
PSO	Quintic	Twister	$4,404105 \cdot 10^{-3}$	$7,535028 \cdot 10^{-2}$	$2,794986 \cdot 10^{-1}$
DE	Quintic	Twister	$1,581993 \cdot 10^{-2}$	$2,038242 \cdot 10^{-1}$	$6,910020 \cdot 10^{-1}$
SOMA	Quintic	Twister	$9,887619 \cdot 10^{-4}$	$1,498316 \cdot 10^{-1}$	$4,790751 \cdot 10^{-1}$
ES	Quintic	Logistic	$7,395597 \cdot 10^{-2}$	$2,356746 \cdot 10^{-1}$	$4,605220 \cdot 10^{-1}$
SA	Quintic	Logistic	$4,681214 \cdot 10^{-2}$	$2,942672 \cdot 10^{-1}$	$7,331245 \cdot 10^{-1}$
PSO	Quintic	Logistic	$2,811095 \cdot 10^{-2}$	$1,631656 \cdot 10^{-1}$	$3,672004 \cdot 10^{-1}$
DE	Quintic	Logistic	$4,982841 \cdot 10^{-2}$	$2,464253 \cdot 10^{-1}$	$8,267810 \cdot 10^{-1}$
SOMA	Quintic	Logistic	$2,849569 \cdot 10^{-2}$	$1,909179 \cdot 10^{-1}$	$7,582497 \cdot 10^{-1}$
ES	Sextic	Twister	$2,013199 \cdot 10^{-2}$	$1,620061 \cdot 10^{-1}$	$3,512800 \cdot 10^{-1}$
SA	Sextic	Twister	$1,382086 \cdot 10^{-2}$	$1,375195 \cdot 10^{-1}$	$3,185936 \cdot 10^{-1}$
PSO	Sextic	Twister	$3,703753 \cdot 10^{-3}$	$6,999829 \cdot 10^{-2}$	$1,584753 \cdot 10^{-1}$
DE	Sextic	Twister	$3,242876 \cdot 10^{-2}$	$1,691297 \cdot 10^{-1}$	$4,028901 \cdot 10^{-1}$
SOMA	Sextic	Twister	$2,886012 \cdot 10^{-2}$	$1,301383 \cdot 10^{-1}$	$3,214312 \cdot 10^{-1}$
ES	Sextic	Logistic	$7,585941 \cdot 10^{-2}$	$2,354852 \cdot 10^{-1}$	$5,239838 \cdot 10^{-1}$
SA	Sextic	Logistic	$6,461101 \cdot 10^{-2}$	$2,594627 \cdot 10^{-1}$	$5,625207 \cdot 10^{-1}$
PSO	Sextic	Logistic	$1,667438 \cdot 10^{-2}$	$1,621630 \cdot 10^{-1}$	$3,184621 \cdot 10^{-1}$
DE	Sextic	Logistic	$5,780111 \cdot 10^{-2}$	$2,709746 \cdot 10^{-1}$	$5,528458 \cdot 10^{-1}$
SOMA	Sextic	Logistic	$6,701062 \cdot 10^{-2}$	$1,833846 \cdot 10^{-1}$	$4,609651 \cdot 10^{-1}$

Tabulka C.1: Výsledné vhodnosti pro algoritmus ES

Algoritmus	Problém	Generátor	Min	Avg	Max
ES	Quintic	Twister	$3,631020 \cdot 10^{-2}$	$1,881189 \cdot 10^{-1}$	$4,474063 \cdot 10^{-1}$
SA	Quintic	Twister	$2,628675 \cdot 10^{-2}$	$1,858399 \cdot 10^{-1}$	$4,369191 \cdot 10^{-1}$
PSO	Quintic	Twister	$8,157133 \cdot 10^{-4}$	$9,047157 \cdot 10^{-2}$	$2,176199 \cdot 10^{-1}$
DE	Quintic	Twister	$2,802304 \cdot 10^{-3}$	$2,050422 \cdot 10^{-1}$	$4,755063 \cdot 10^{-1}$
SOMA	Quintic	Twister	$6,069250 \cdot 10^{-2}$	$1,959449 \cdot 10^{-1}$	$4,406146 \cdot 10^{-1}$
ES	Quintic	Logistic	$7,859740 \cdot 10^{-2}$	$2,526808 \cdot 10^{-1}$	$5,314087 \cdot 10^{-1}$
SA	Quintic	Logistic	$1,152824 \cdot 10^{-1}$	$3,333417 \cdot 10^{-1}$	$1,018202 \cdot 10^{0}$
PSO	Quintic	Logistic	$1,338242 \cdot 10^{-2}$	$2,162447 \cdot 10^{-1}$	$5,468906 \cdot 10^{-1}$
DE	Quintic	Logistic	$7,233028 \cdot 10^{-2}$	$2,909394 \cdot 10^{-1}$	$7,768615 \cdot 10^{-1}$
SOMA	Quintic	Logistic	$7,699697 \cdot 10^{-2}$	$2,118185 \cdot 10^{-1}$	$4,097938 \cdot 10^{-1}$
ES	Sextic	Twister	$1,810522 \cdot 10^{-2}$	$1,832978 \cdot 10^{-1}$	$4,477897 \cdot 10^{-1}$
SA	Sextic	Twister	$2,748267 \cdot 10^{-2}$	$1,690862 \cdot 10^{-1}$	$4,715378 \cdot 10^{-1}$
PSO	Sextic	Twister	$4,440892 \cdot 10^{-16}$	$9,127263 \cdot 10^{-2}$	$2,601946 \cdot 10^{-1}$
DE	Sextic	Twister	$4,082605 \cdot 10^{-2}$	$2,274636 \cdot 10^{-1}$	$6,068369 \cdot 10^{-1}$
SOMA	Sextic	Twister	$6,204358 \cdot 10^{-3}$	$1,632882 \cdot 10^{-1}$	$3,303328 \cdot 10^{-1}$
ES	Sextic	Logistic	$7,027073 \cdot 10^{-2}$	$2,658558 \cdot 10^{-1}$	$8,019849 \cdot 10^{-1}$
SA	Sextic	Logistic	$7,959370 \cdot 10^{-2}$	$2,994236 \cdot 10^{-1}$	$5,619098 \cdot 10^{-1}$
PSO	Sextic	Logistic	$6,282266 \cdot 10^{-2}$	$2,045140 \cdot 10^{-1}$	$4,297889 \cdot 10^{-1}$
DE	Sextic	Logistic	$7,909567 \cdot 10^{-2}$	$2,777427 \cdot 10^{-1}$	$5,511874 \cdot 10^{-1}$
SOMA	Sextic	Logistic	$6,732079 \cdot 10^{-2}$	$2,143204 \cdot 10^{-1}$	$4,291408 \cdot 10^{-1}$

Tabulka C.2: Výsledné vhodnosti pro algoritmus SA

Algoritmus	Problém	Generátor	Min	Avg	Max
ES	Quintic	Twister	$1,114106 \cdot 10^{-1}$	$3,966371 \cdot 10^{-1}$	$1,064116 \cdot 10^{0}$
SA	Quintic	Twister	$1,741304 \cdot 10^{-2}$	$4,081749 \cdot 10^{-1}$	$1,132734 \cdot 10^{0}$
PSO	Quintic	Twister	$4,440892 \cdot 10^{-16}$	$1,827600 \cdot 10^{-1}$	$5,704516 \cdot 10^{-1}$
DE	Quintic	Twister	$6,467250 \cdot 10^{-2}$	$4,882525 \cdot 10^{-1}$	$1,417831 \cdot 10^{0}$
SOMA	Quintic	Twister	$7,912938 \cdot 10^{-2}$	$3,545132 \cdot 10^{-1}$	$9,665829 \cdot 10^{-1}$
ES	Quintic	Logistic	$1,345586 \cdot 10^{-2}$	$1,879449 \cdot 10^{-1}$	$4,757598 \cdot 10^{-1}$
SA	Quintic	Logistic	$7,115763 \cdot 10^{-2}$	$2,474832 \cdot 10^{-1}$	$6,149282 \cdot 10^{-1}$
PSO	Quintic	Logistic	$6,661338 \cdot 10^{-16}$	$1,577695 \cdot 10^{-1}$	$5,243795 \cdot 10^{-1}$
DE	Quintic	Logistic	$4,107135 \cdot 10^{-2}$	$2,751864 \cdot 10^{-1}$	$6,925307 \cdot 10^{-1}$
SOMA	Quintic	Logistic	$8,881784 \cdot 10^{-16}$	$1,462499 \cdot 10^{-1}$	$4,337883 \cdot 10^{-1}$
ES	Sextic	Twister	$6,761927 \cdot 10^{-2}$	$2,919158 \cdot 10^{-1}$	$1,094095 \cdot 10^{0}$
SA	Sextic	Twister	$3,653574 \cdot 10^{-2}$	$2,938955 \cdot 10^{-1}$	$8,950967 \cdot 10^{-1}$
PSO	Sextic	Twister	$6,038816 \cdot 10^{-4}$	$1,714961 \cdot 10^{-1}$	$5,206356 \cdot 10^{-1}$
DE	Sextic	Twister	$3,907452 \cdot 10^{-3}$	$3,480053 \cdot 10^{-1}$	$7,971882 \cdot 10^{-1}$
SOMA	Sextic	Twister	$5,606501 \cdot 10^{-2}$	$2,516071 \cdot 10^{-1}$	$8,983202 \cdot 10^{-1}$
ES	Sextic	Logistic	$2,407037 \cdot 10^{-2}$	$1,936008 \cdot 10^{-1}$	$5,131898 \cdot 10^{-1}$
SA	Sextic	Logistic	$5,152172 \cdot 10^{-2}$	$2,131608 \cdot 10^{-1}$	$5,533837 \cdot 10^{-1}$
PSO	Sextic	Logistic	$4,701266 \cdot 10^{-2}$	$1,444130 \cdot 10^{-1}$	$4,620332 \cdot 10^{-1}$
DE	Sextic	Logistic	$1,524099 \cdot 10^{-2}$	$2,362077 \cdot 10^{-1}$	$7,368225 \cdot 10^{-1}$
SOMA	Sextic	Logistic	$3,510573 \cdot 10^{-4}$	$1,506121 \cdot 10^{-1}$	$4,710984 \cdot 10^{-1}$

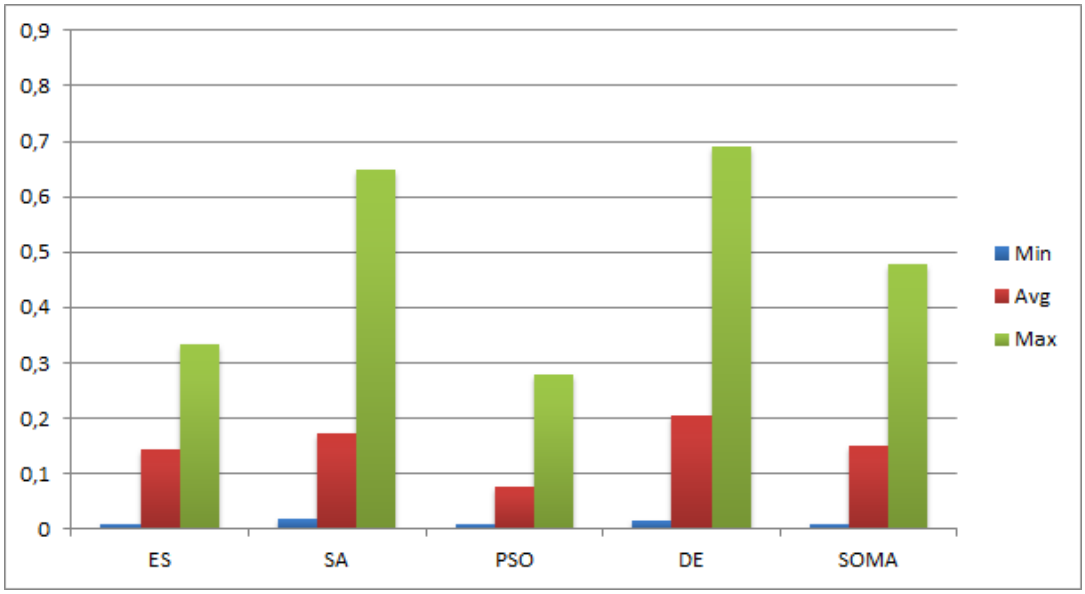
Tabulka C.3: Výsledné vhodnosti pro algoritmus PSO

Algoritmus	Problém	Generátor	Min	Avg	Max
ES	Quintic	Twister	$8,141989 \cdot 10^{-2}$	$2,765640 \cdot 10^{-1}$	$6,124803 \cdot 10^{-1}$
SA	Quintic	Twister	$1,326776 \cdot 10^{-2}$	$2,897637 \cdot 10^{-1}$	$6,862502 \cdot 10^{-1}$
PSO	Quintic	Twister	$6,210946 \cdot 10^{-3}$	$1,651474 \cdot 10^{-1}$	$3,963387 \cdot 10^{-1}$
DE	Quintic	Twister	$7,526402 \cdot 10^{-2}$	$3,581147 \cdot 10^{-1}$	$7,064366 \cdot 10^{-1}$
SOMA	Quintic	Twister	$7,492046 \cdot 10^{-2}$	$2,683223 \cdot 10^{-1}$	$6,907390 \cdot 10^{-1}$
ES	Quintic	Logistic	$1,180713 \cdot 10^{-1}$	$4,346338 \cdot 10^{-1}$	$1,482279 \cdot 10^{0}$
SA	Quintic	Logistic	$7,306709 \cdot 10^{-2}$	$3,751464 \cdot 10^{-1}$	$8,450164 \cdot 10^{-1}$
PSO	Quintic	Logistic	$8,361479 \cdot 10^{-2}$	$2,679155 \cdot 10^{-1}$	$7,780357 \cdot 10^{-1}$
DE	Quintic	Logistic	$1,533089 \cdot 10^{-1}$	$4,186863 \cdot 10^{-1}$	$8,239361 \cdot 10^{-1}$
SOMA	Quintic	Logistic	$5,621102 \cdot 10^{-2}$	$2,623898 \cdot 10^{-1}$	$7,855465 \cdot 10^{-1}$
ES	Sextic	Twister	$6,555551 \cdot 10^{-2}$	$2,499120 \cdot 10^{-1}$	$6,213965 \cdot 10^{-1}$
SA	Sextic	Twister	$5,714061 \cdot 10^{-2}$	$2,153398 \cdot 10^{-1}$	$4,586903 \cdot 10^{-1}$
PSO	Sextic	Twister	$5,626506 \cdot 10^{-3}$	$1,476360 \cdot 10^{-1}$	$3,549581 \cdot 10^{-1}$
DE	Sextic	Twister	$4,714791 \cdot 10^{-2}$	$3,033165 \cdot 10^{-1}$	$8,527469 \cdot 10^{-1}$
SOMA	Sextic	Twister	$3,807259 \cdot 10^{-2}$	$2,465887 \cdot 10^{-1}$	$5,334007 \cdot 10^{-1}$
ES	Sextic	Logistic	$7,832922 \cdot 10^{-2}$	$3,403758 \cdot 10^{-1}$	$5,718663 \cdot 10^{-1}$
SA	Sextic	Logistic	$8,628651 \cdot 10^{-2}$	$3,717240 \cdot 10^{-1}$	$8,610967 \cdot 10^{-1}$
PSO	Sextic	Logistic	$9,085486 \cdot 10^{-2}$	$2,687028 \cdot 10^{-1}$	$5,069571 \cdot 10^{-1}$
DE	Sextic	Logistic	$8,739261 \cdot 10^{-2}$	$3,983626 \cdot 10^{-1}$	$7,030127 \cdot 10^{-1}$
SOMA	Sextic	Logistic	$8,469083 \cdot 10^{-2}$	$2,784596 \cdot 10^{-1}$	$8,432661 \cdot 10^{-1}$

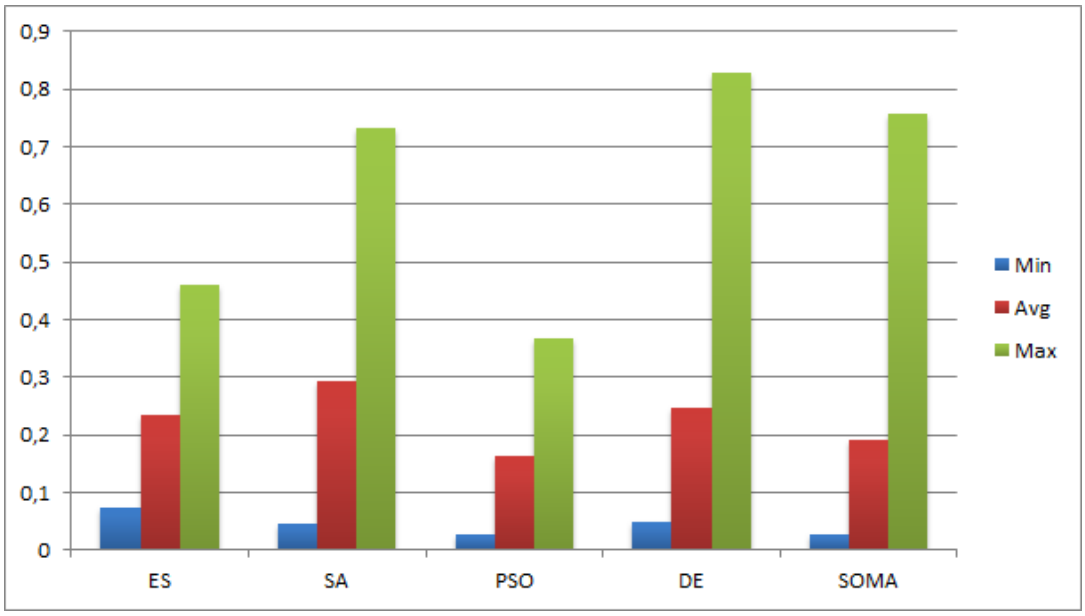
Tabulka C.4: Výsledné vhodnosti pro algoritmus DE

Algoritmus	Problém	Generátor	Min	Avg	Max
ES	Quintic	Twister	$7,239620 \cdot 10^{-3}$	$7,793688 \cdot 10^{-2}$	$3,908374 \cdot 10^{-1}$
SA	Quintic	Twister	$4,573641 \cdot 10^{-3}$	$7,341193 \cdot 10^{-2}$	$2,411200 \cdot 10^{-1}$
PSO	Quintic	Twister	$6,661338 \cdot 10^{-16}$	$2,896303 \cdot 10^{-2}$	$1,128802 \cdot 10^{-1}$
DE	Quintic	Twister	$1,487343 \cdot 10^{-2}$	$1,072624 \cdot 10^{-1}$	$3,999587 \cdot 10^{-1}$
SOMA	Quintic	Twister	$2,878224 \cdot 10^{-3}$	$6,141635 \cdot 10^{-2}$	$1,686350 \cdot 10^{-1}$
ES	Quintic	Logistic	$1,423070 \cdot 10^{-2}$	$1,133331 \cdot 10^{-1}$	$3,573011 \cdot 10^{-1}$
SA	Quintic	Logistic	$4,125048 \cdot 10^{-2}$	$1,467077 \cdot 10^{-1}$	$3,702667 \cdot 10^{-1}$
PSO	Quintic	Logistic	$1,008551 \cdot 10^{-3}$	$7,077869 \cdot 10^{-2}$	$1,489960 \cdot 10^{-1}$
DE	Quintic	Logistic	$8,707820 \cdot 10^{-3}$	$1,165570 \cdot 10^{-1}$	$3,387989 \cdot 10^{-1}$
SOMA	Quintic	Logistic	$1,689309 \cdot 10^{-2}$	$8,068816 \cdot 10^{-2}$	$2,545417 \cdot 10^{-1}$
ES	Sextic	Twister	$5,448487 \cdot 10^{-3}$	$7,267151 \cdot 10^{-2}$	$3,159944 \cdot 10^{-1}$
SA	Sextic	Twister	$1,565532 \cdot 10^{-2}$	$7,109185 \cdot 10^{-2}$	$2,220822 \cdot 10^{-1}$
PSO	Sextic	Twister	$1,853924 \cdot 10^{-3}$	$2,854710 \cdot 10^{-2}$	$1,001054 \cdot 10^{-1}$
DE	Sextic	Twister	$1,374556 \cdot 10^{-2}$	$8,419237 \cdot 10^{-2}$	$2,099822 \cdot 10^{-1}$
SOMA	Sextic	Twister	$4,790133 \cdot 10^{-3}$	$5,407930 \cdot 10^{-2}$	$1,639934 \cdot 10^{-1}$
ES	Sextic	Logistic	$2,703962 \cdot 10^{-2}$	$1,129918 \cdot 10^{-1}$	$2,370036 \cdot 10^{-1}$
SA	Sextic	Logistic	$2,102164 \cdot 10^{-2}$	$1,354844 \cdot 10^{-1}$	$3,289218 \cdot 10^{-1}$
PSO	Sextic	Logistic	$2,269274 \cdot 10^{-2}$	$8,976429 \cdot 10^{-2}$	$3,575043 \cdot 10^{-1}$
DE	Sextic	Logistic	$2,332323 \cdot 10^{-2}$	$1,358028 \cdot 10^{-1}$	$3,425571 \cdot 10^{-1}$
SOMA	Sextic	Logistic	$6,803753 \cdot 10^{-3}$	$9,660472 \cdot 10^{-2}$	$2,223559 \cdot 10^{-1}$

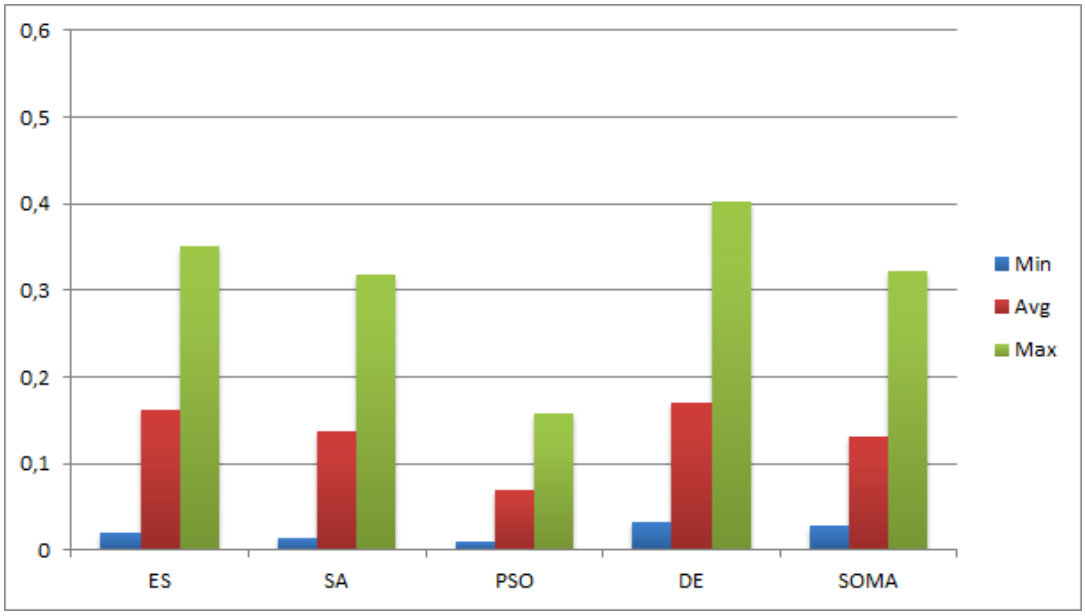
Tabulka C.5: Výsledné vhodnosti pro algoritmus SOMA



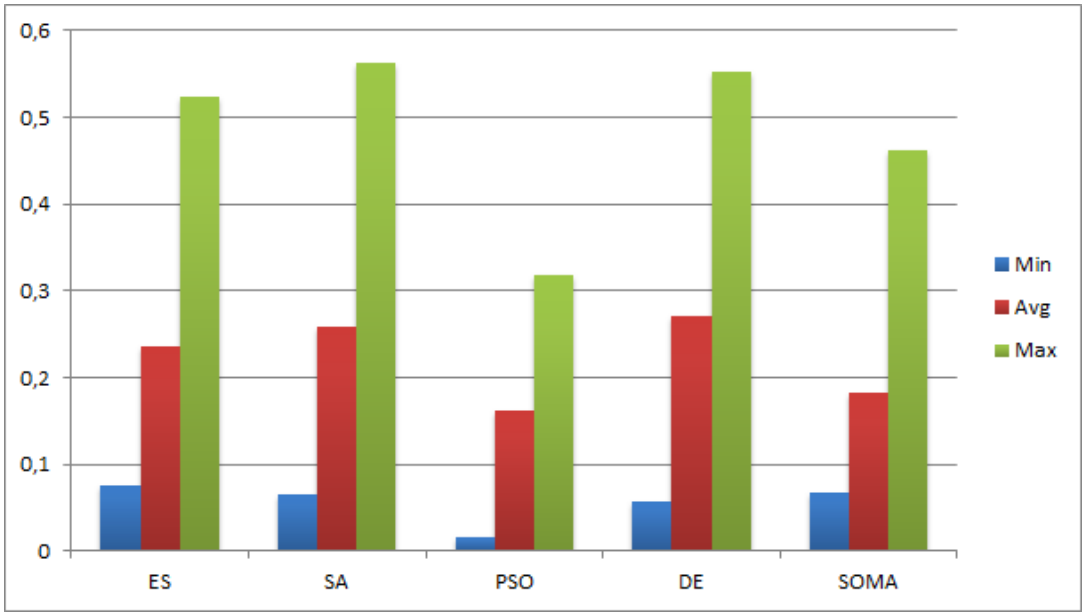
Obrázek C.1: Vhodnosti pro ES, Quintic problém a generátor MersenneTwister



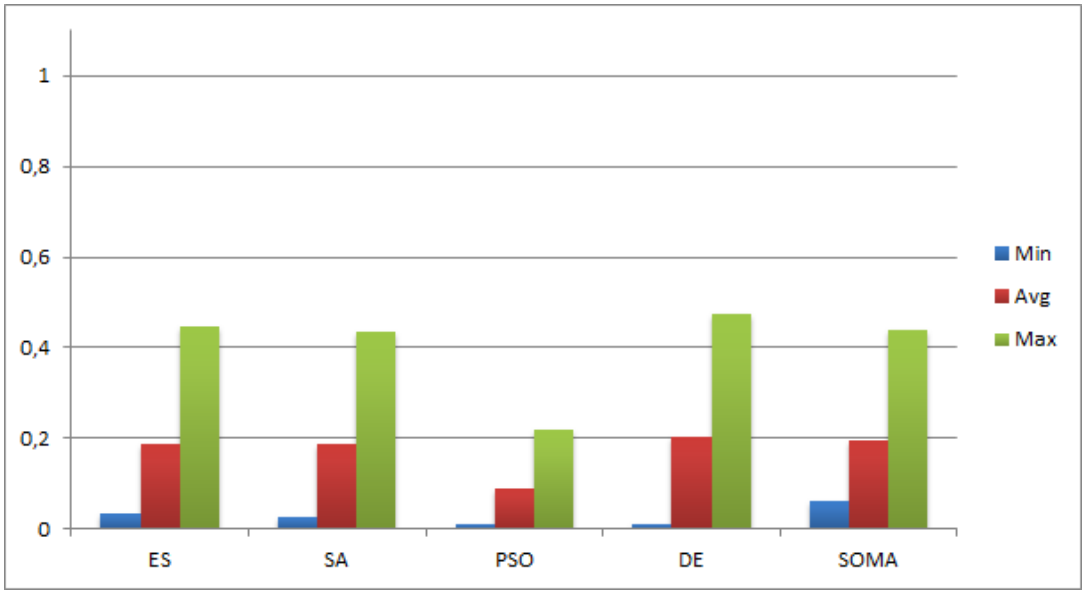
Obrázek C.2: Vhodnosti pro ES, Quintic problém a generátor LogisticMap



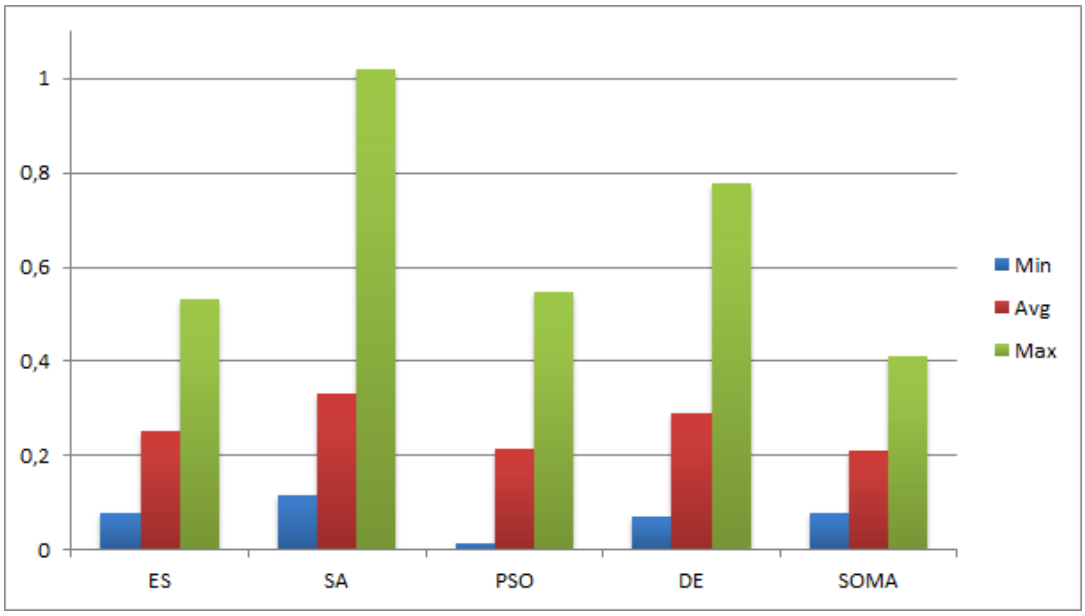
Obrázek C.3: Vhodnosti pro ES, Sextic problém a generátor MersenneTwister



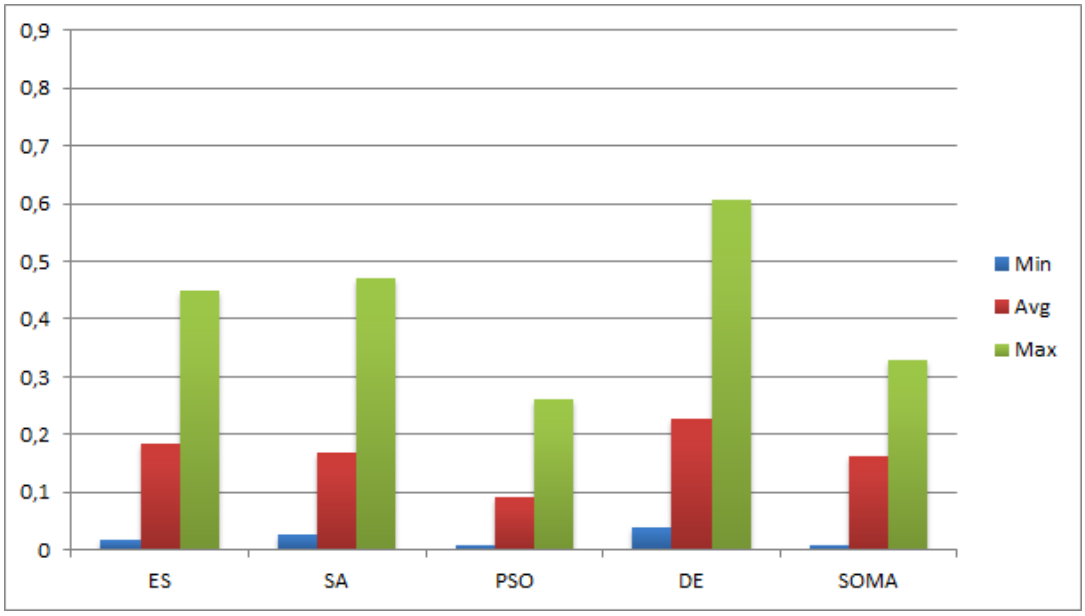
Obrázek C.4: Vhodnosti pro ES, Sextic problém a generátor LogisticMap



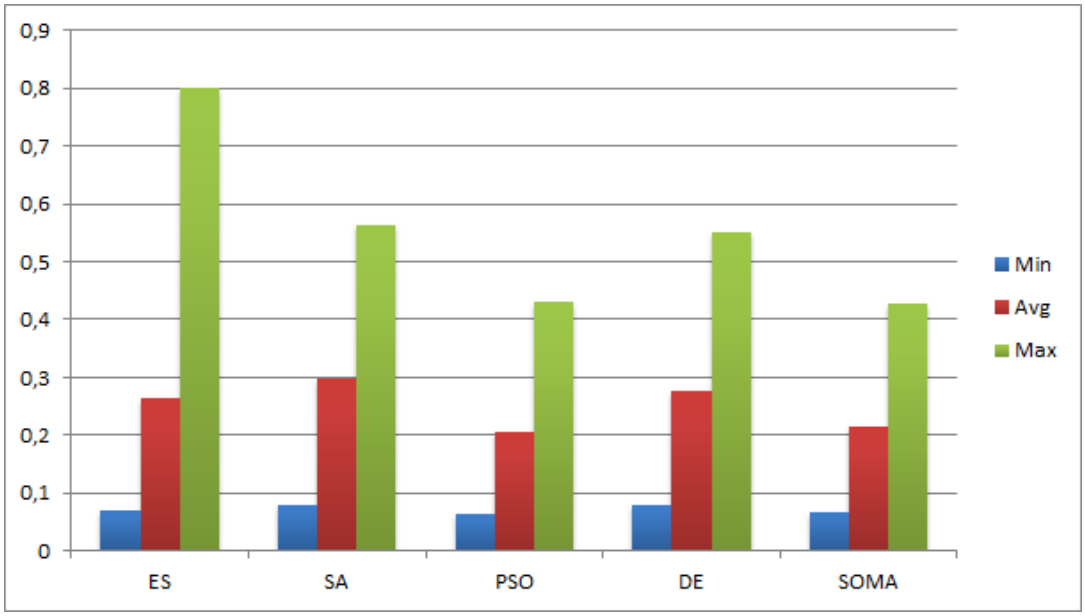
Obrázek C.5: Vhodnosti pro SA, Quintic problém a generátor MersenneTwister



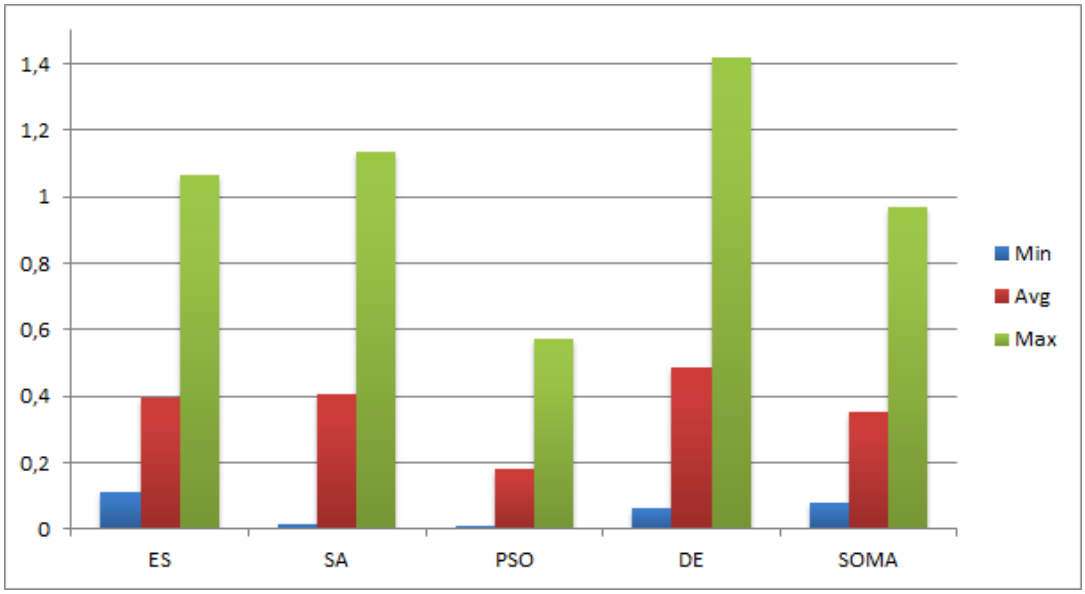
Obrázek C.6: Vhodnosti pro SA, Quintic problém a generátor LogisticMap



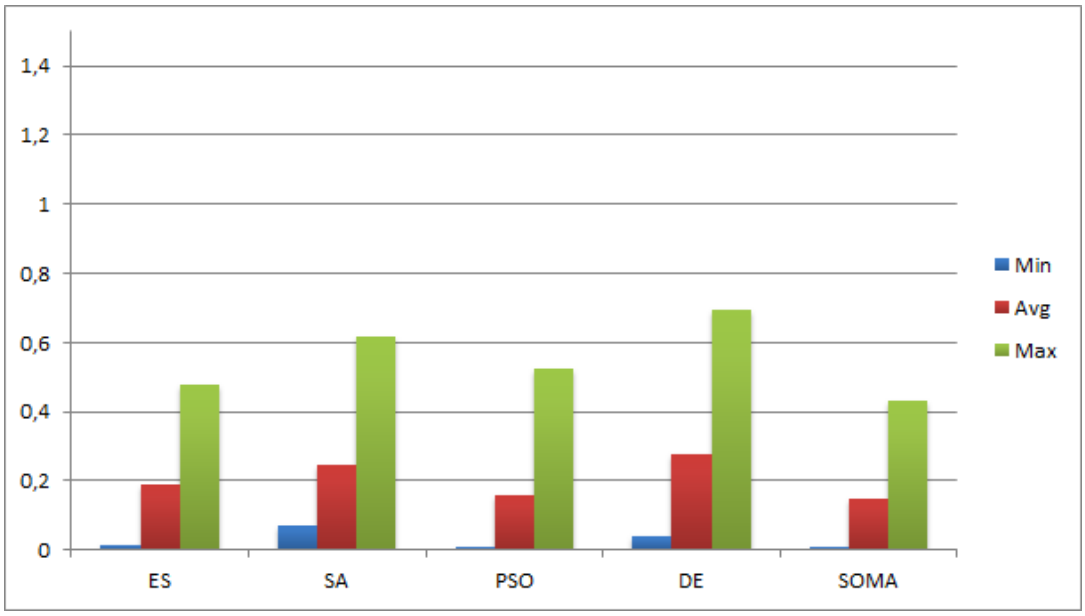
Obrázek C.7: Vhodnosti pro SA, Sextic problém a generátor MersenneTwister



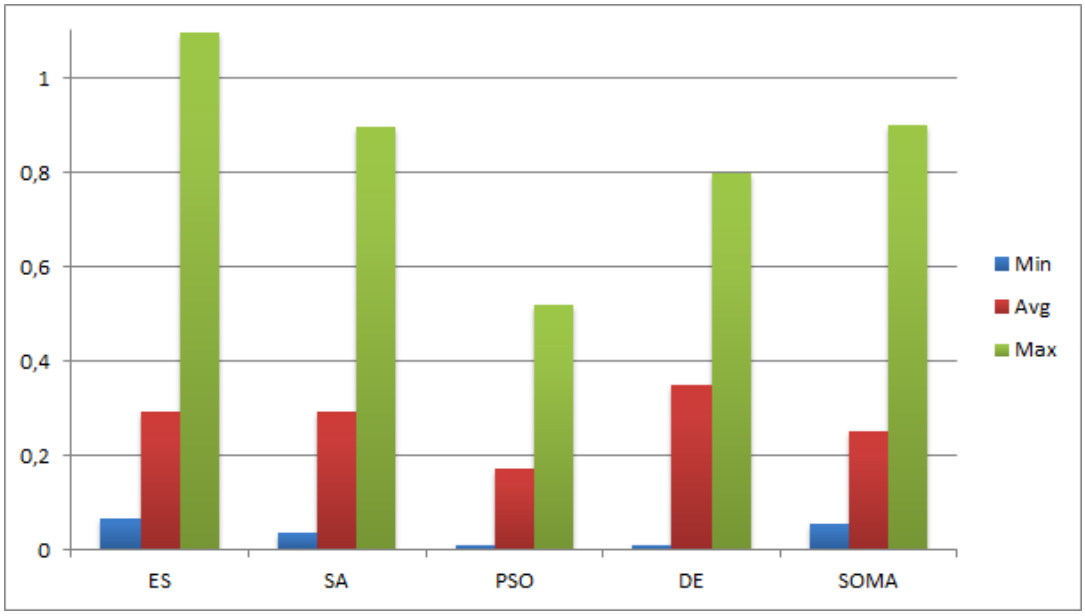
Obrázek C.8: Vhodnosti pro SA, Sextic problém a generátor LogisticMap



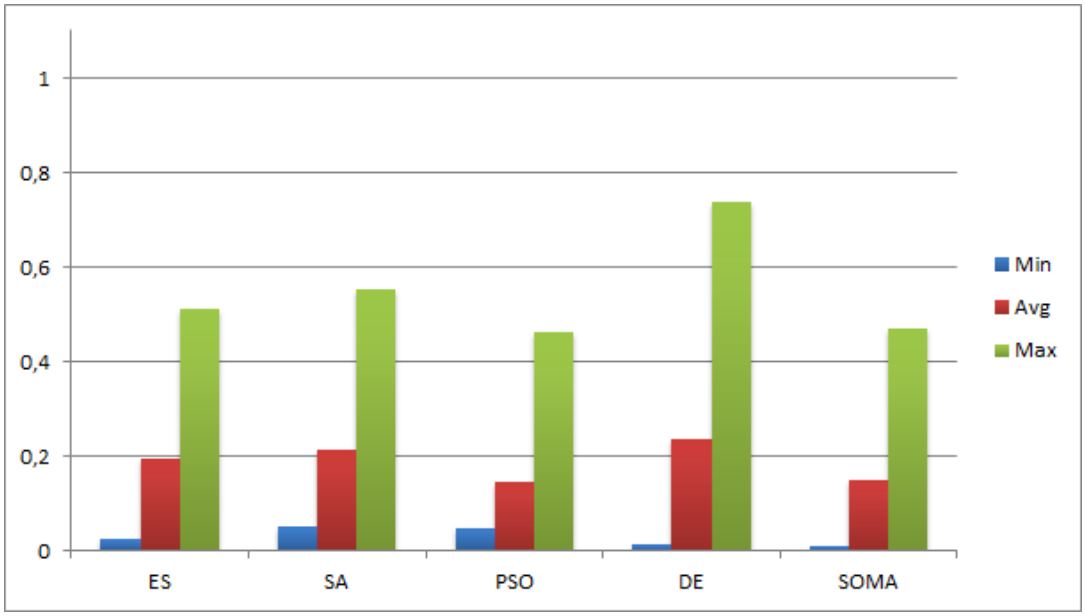
Obrázek C.9: Vhodnosti pro PSO, Quintic problém a generátor MersenneTwister



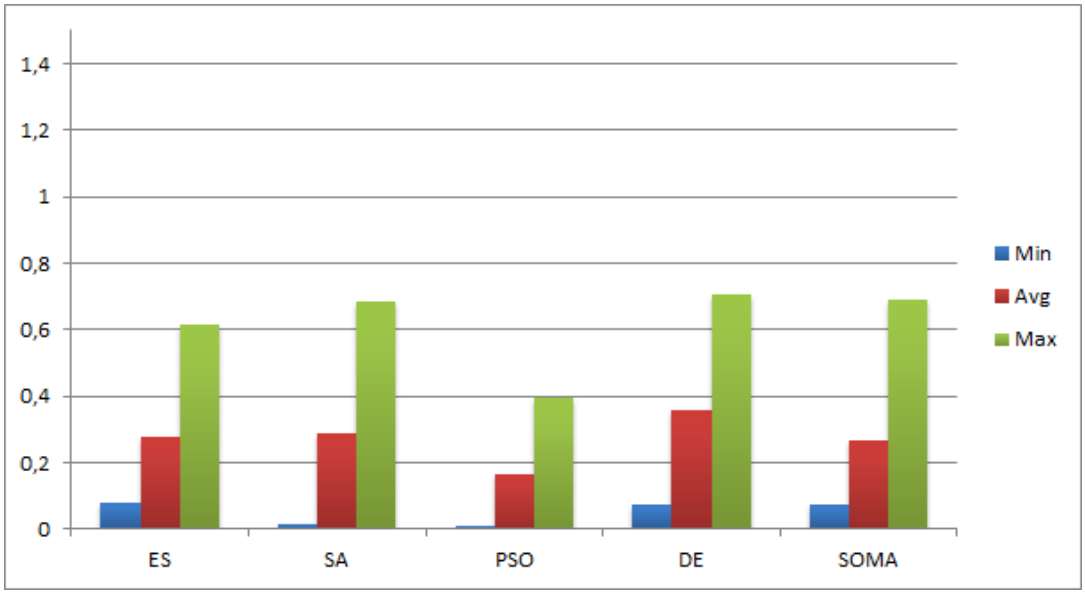
Obrázek C.10: Vhodnosti pro PSO, Quintic problém a generátor LogisticMap



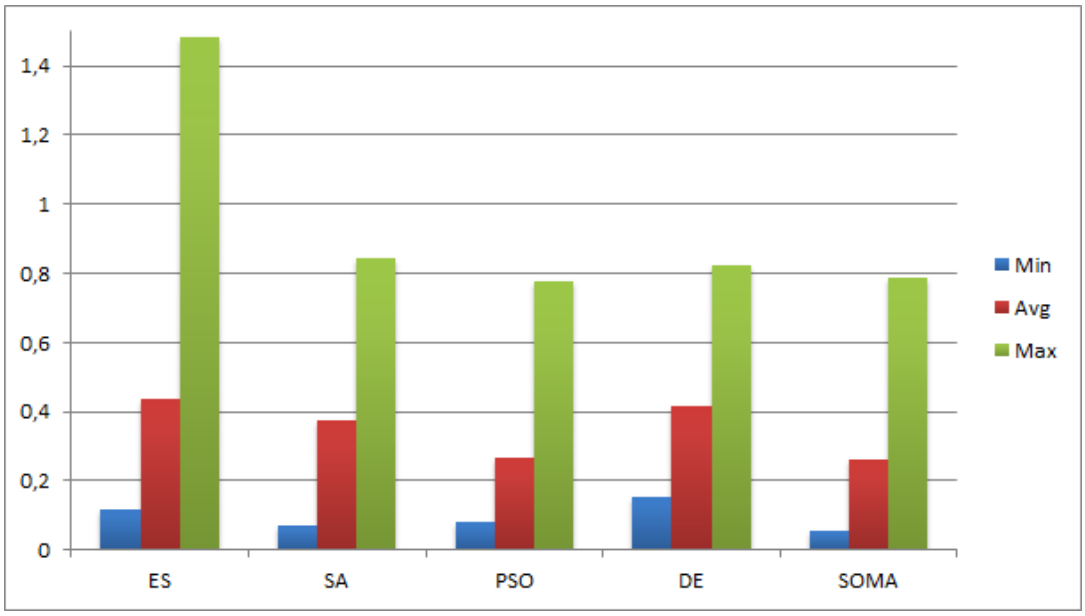
Obrázek C.11: Vhodnosti pro PSO, Sextic problém a generátor MersenneTwister



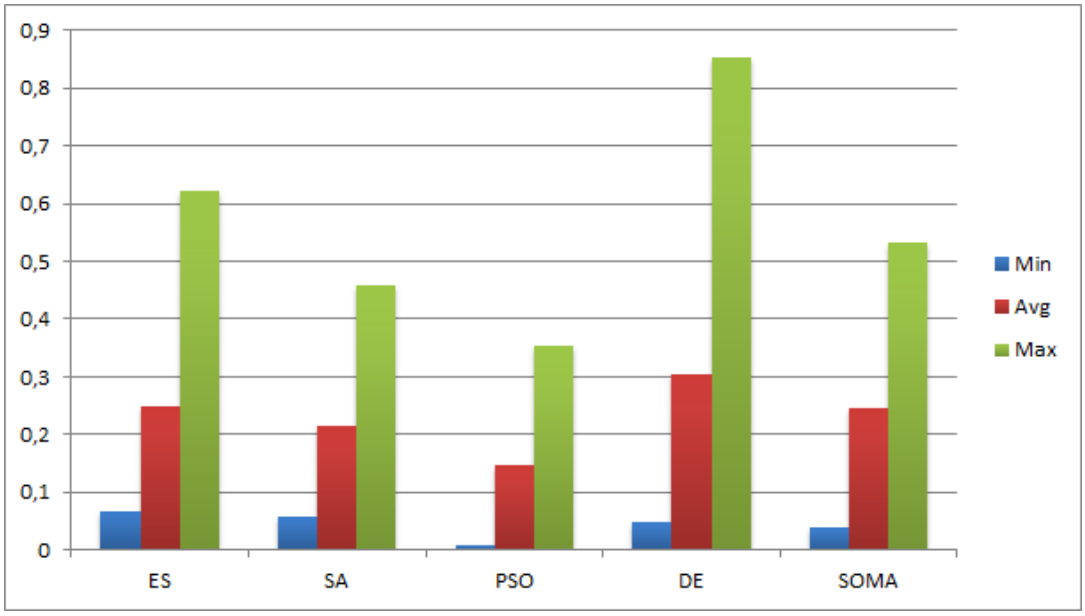
Obrázek C.12: Vhodnosti pro PSO, Sextic problém a generátor LogisticMap



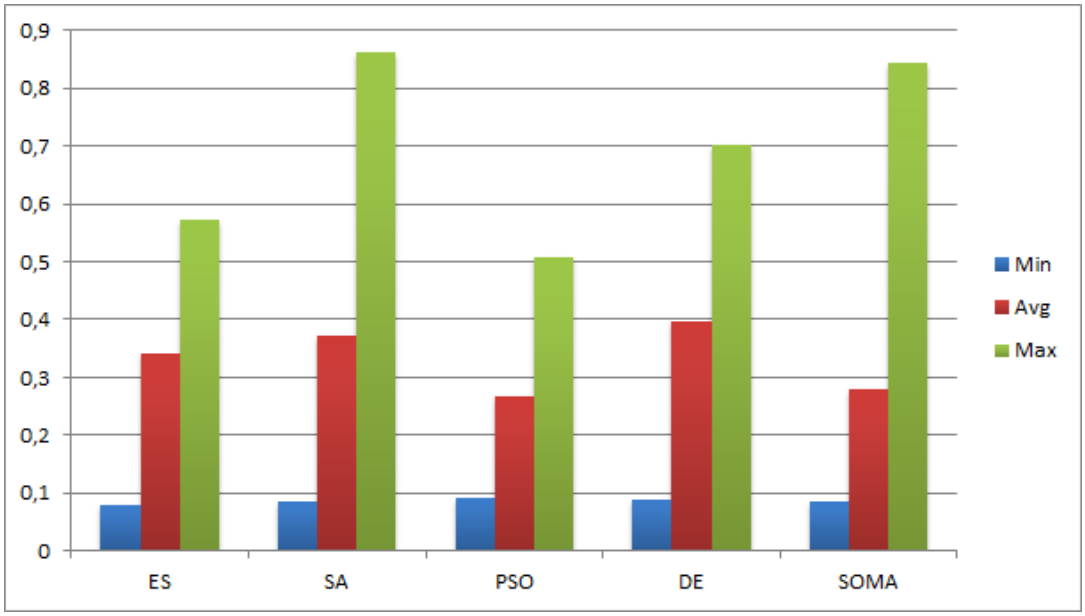
Obrázek C.13: Vhodnosti pro DE, Quintic problém a generátor MersenneTwister



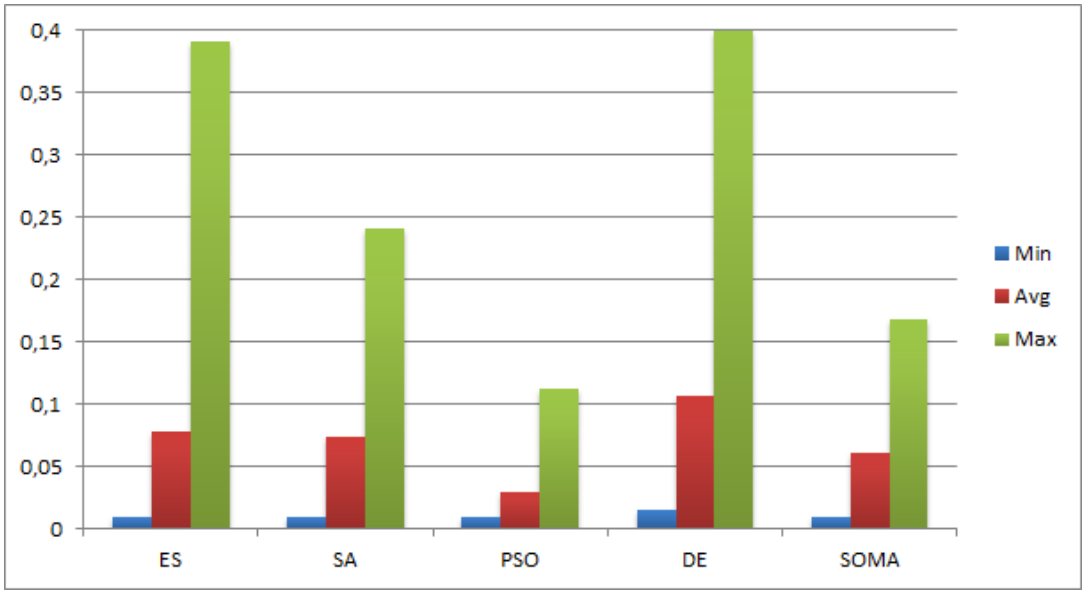
Obrázek C.14: Vhodnosti pro DE, Quintic problém a generátor LogisticMap



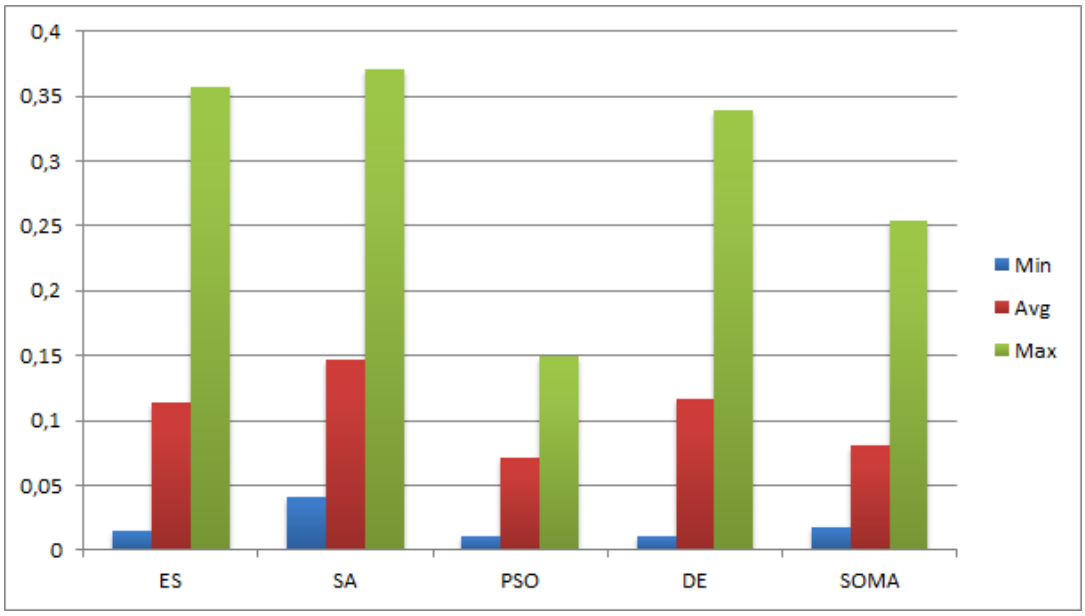
Obrázek C.15: Vhodnosti pro DE, Sextic problém a generátor MersenneTwister



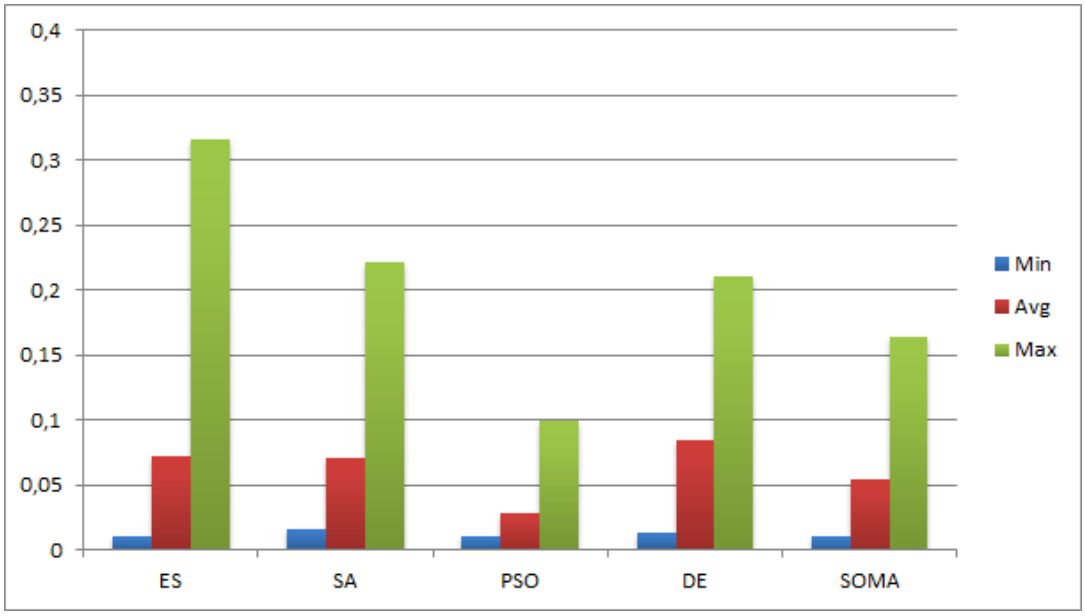
Obrázek C.16: Vhodnosti pro DE, Sextic problém a generátor LogisticMap



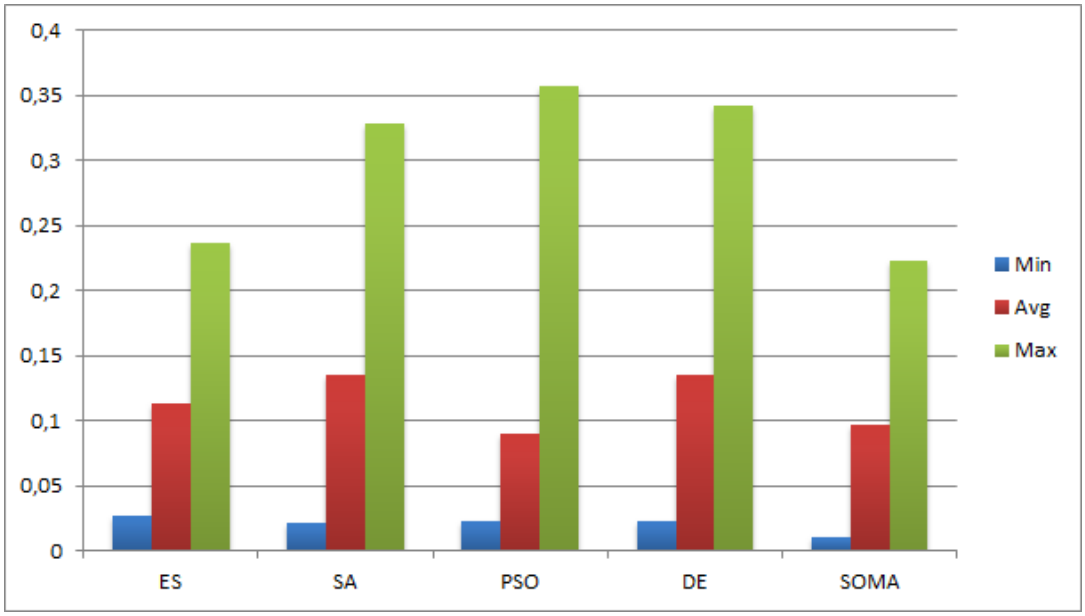
Obrázek C.17: Vhodnosti pro SOMA, Quintic problém a generátor MersenneTwister



Obrázek C.18: Vhodnosti pro SOMA, Quintic problém a generátor LogisticMap

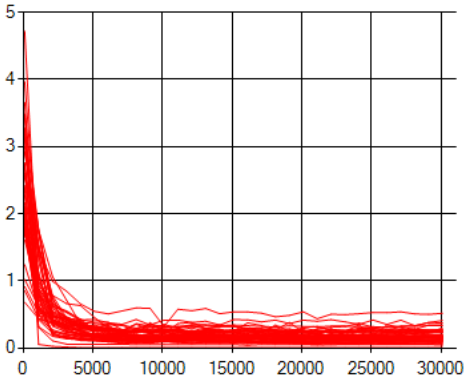


Obrázek C.19: Vhodnosti pro SOMA, Sextic problém a generátor MersenneTwister

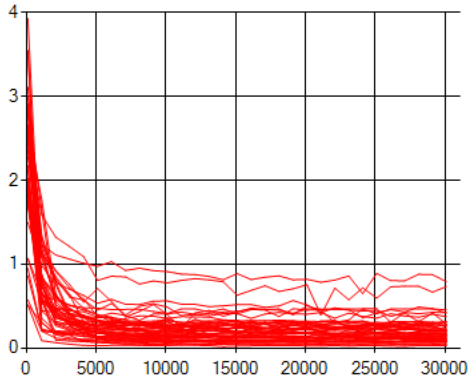


Obrázek C.20: Vhodnosti pro SOMA, Sextic problém a generátor LogisticMap

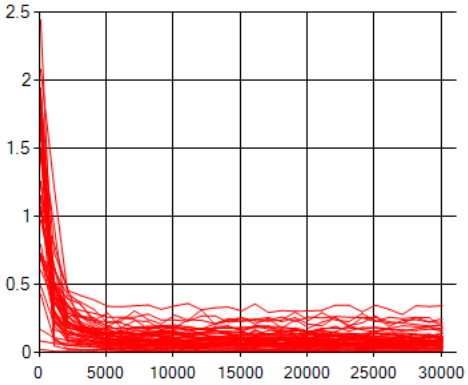
D Průběhy evolučních algoritmů



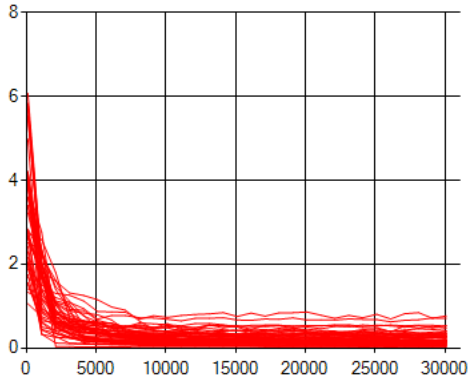
ES_ES_quintic_twister



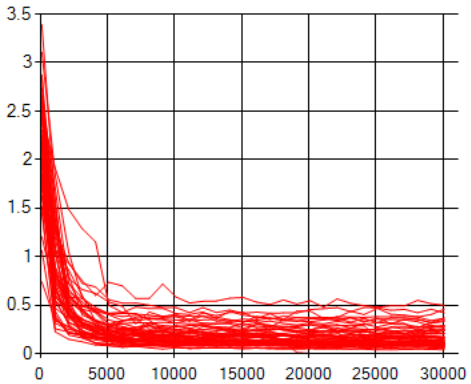
ES_SA_quintic_twister



ES_PSO_quintic_twister

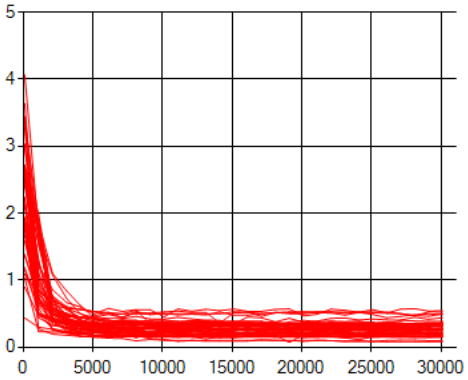


ES_DE_quintic_twister

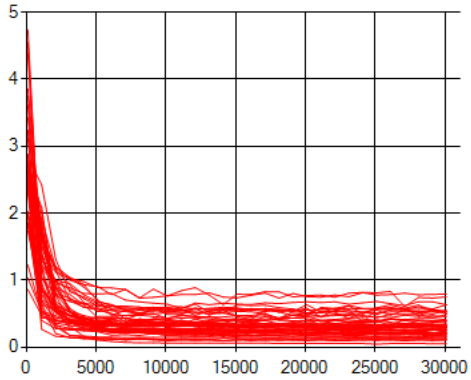


ES_SOMA_quintic_twister

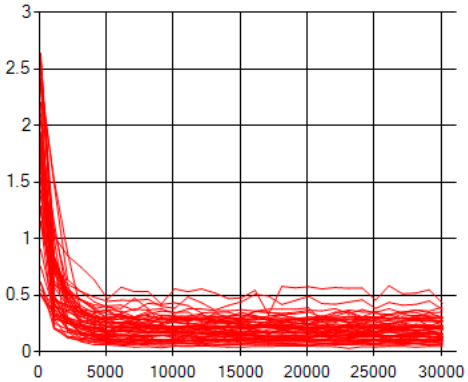
Obrázek D.1: Průběh ES – Quintic problém – MersenneTwister



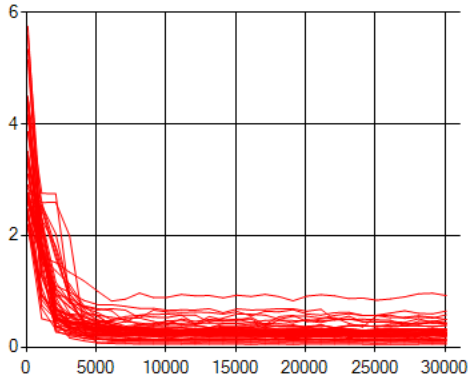
ES_ES_quintic_logistic



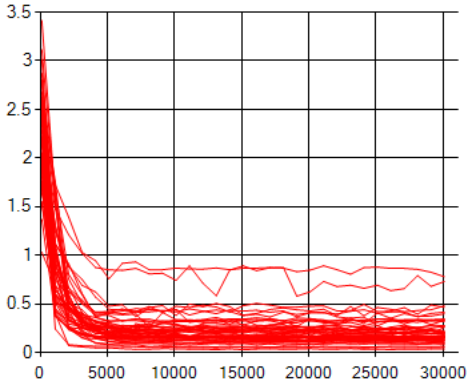
ES_SA_quintic_logistic



ES_PSO_quintic_logistic

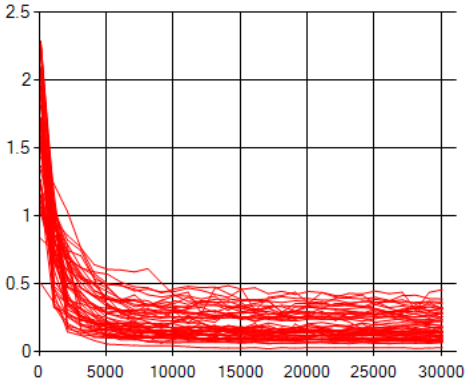


ES_DE_quintic_logistic

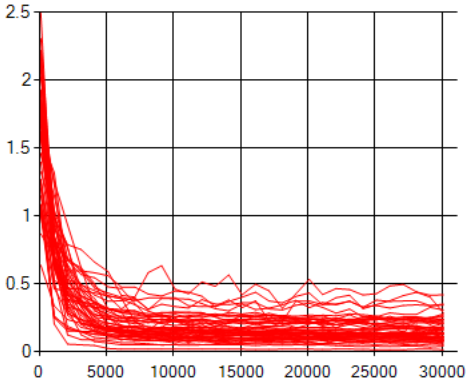


ES_SOMA_quintic_logistic

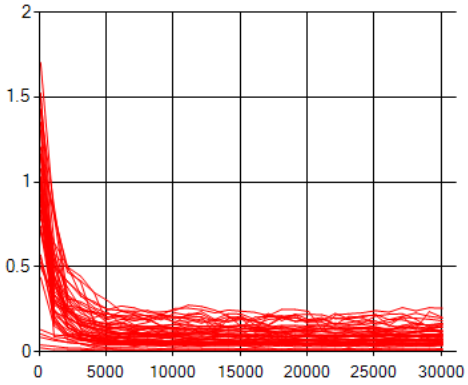
Obrázek D.2: Průběh ES – Quintic problém – LogisticMap



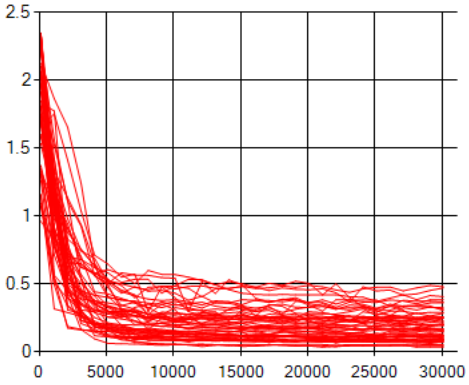
ES_ES_sextic_twister



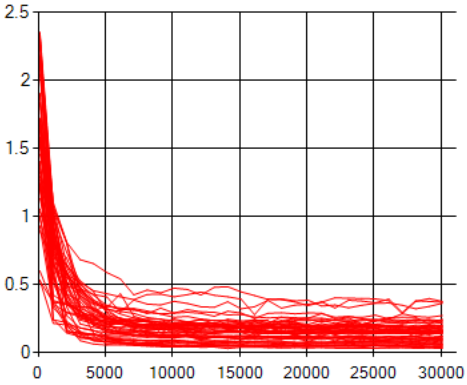
ES_SA_sextic_twister



ES_PSO_sextic_twister

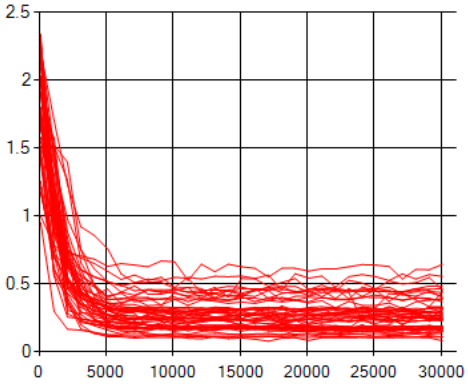


ES_DE_sextic_twister

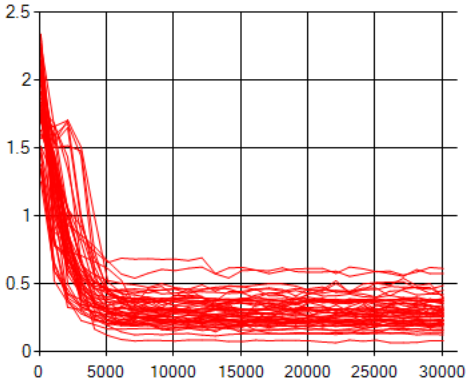


ES_SOMA_sextic_twister

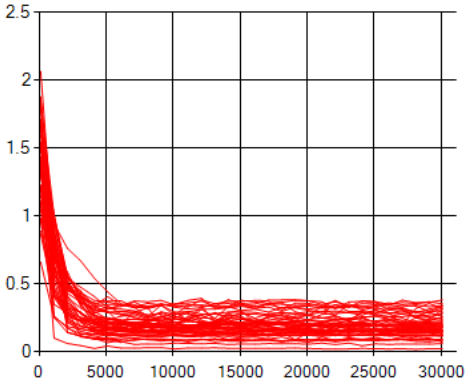
Obrázek D.3: Průběh ES – Sextic problém – MersenneTwister



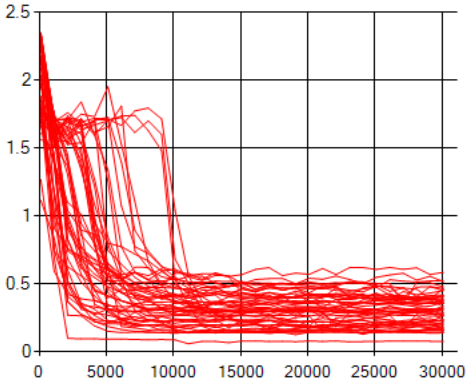
ES_ES_sextic_logistic



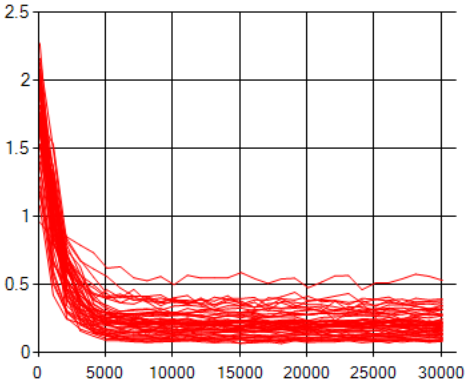
ES_SA_sextic_logistic



ES_PSO_sextic_logistic

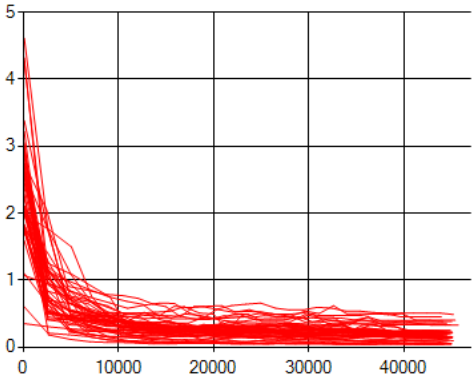


ES_DE_sextic_logistic

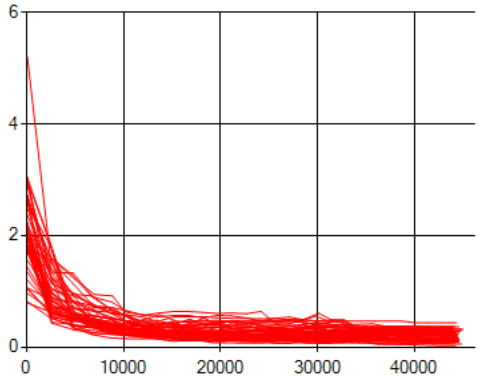


ES_SOMA_sextic_logistic

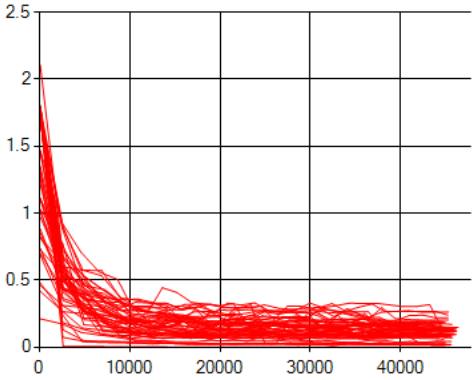
Obrázek D.4: Průběh ES – Sextic problém – LogisticMap



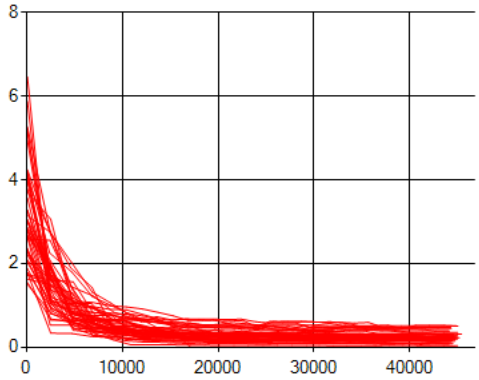
SA_ES_quintic_twister



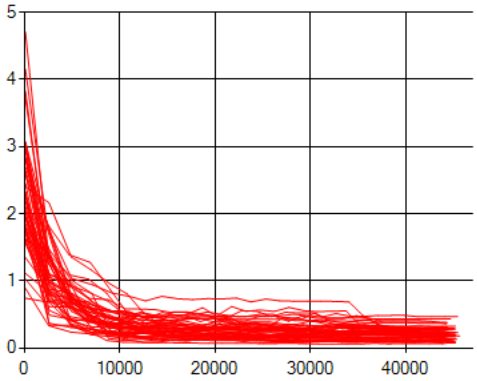
SA_SA_quintic_twister



SA_PSO_quintic_twister

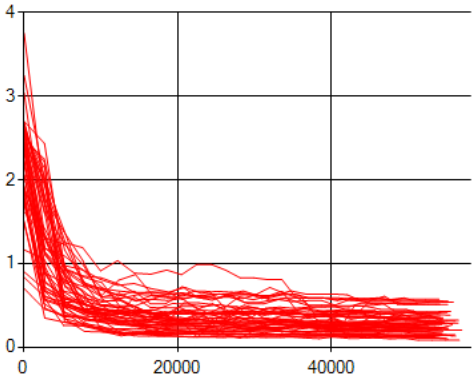


SA_DE_quintic_twister

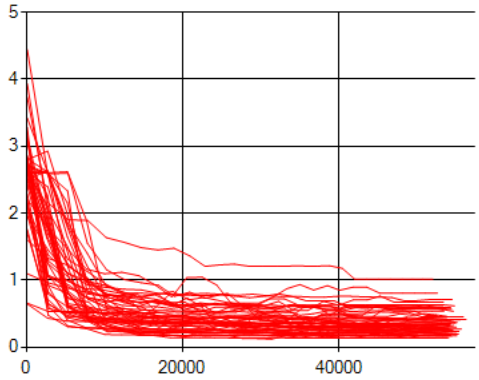


SA_SOMA_quintic_twister

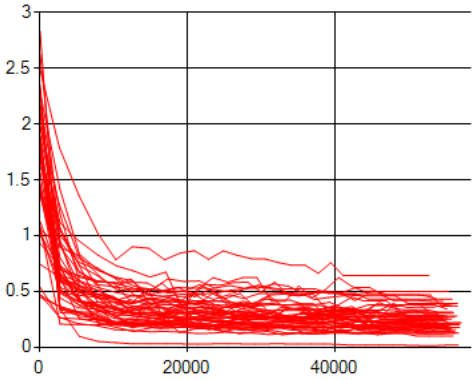
Obrázek D.5: Průběh SA – Quintic problém – MersenneTwister



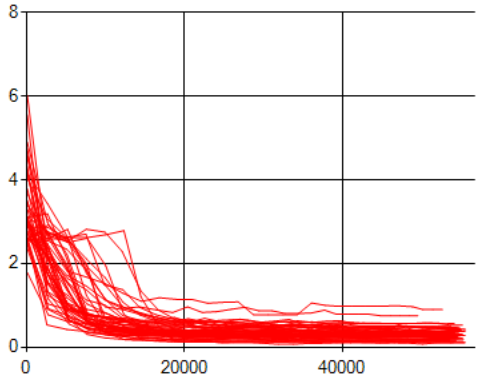
SA_ES_quintic_logistic



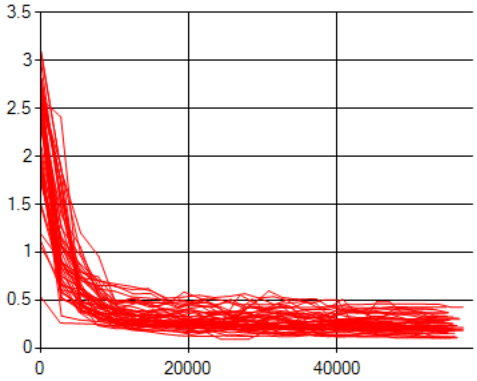
SA_SA_quintic_logistic



SA_PSO_quintic_logistic

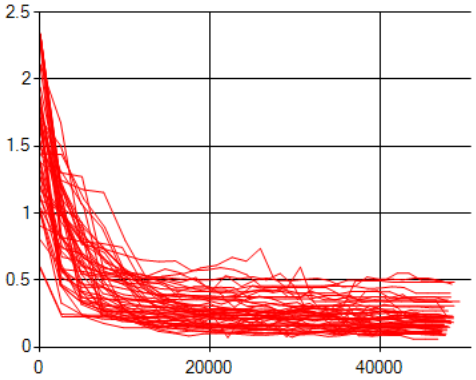


SA_DE_quintic_logistic

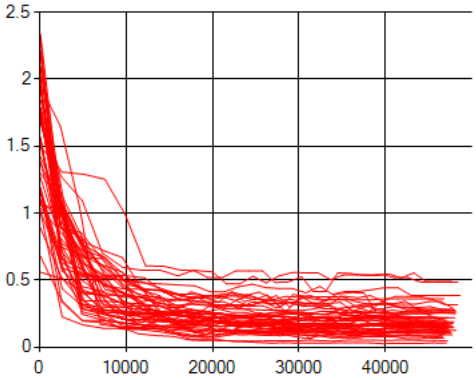


SA_SOMA_quintic_logistic

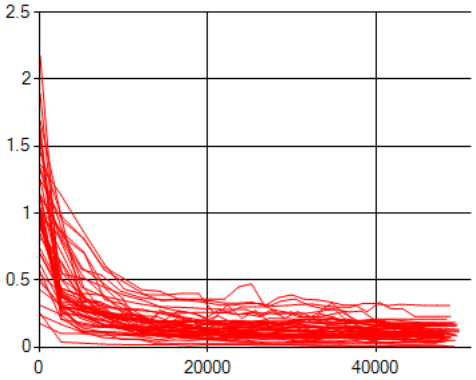
Obrázek D.6: Průběh SA – Quintic problém – LogisticMap



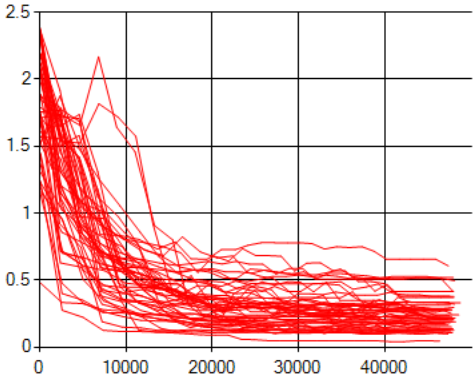
SA_ES_sextic_twister



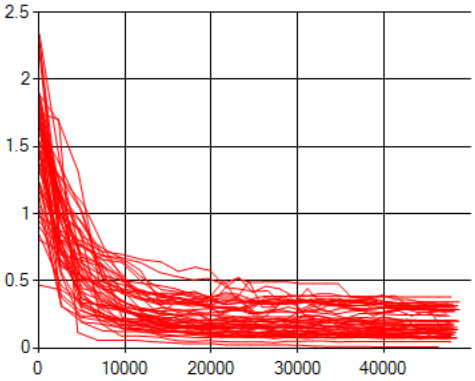
SA_SA_sextic_twister



SA_PSO_sextic_twister

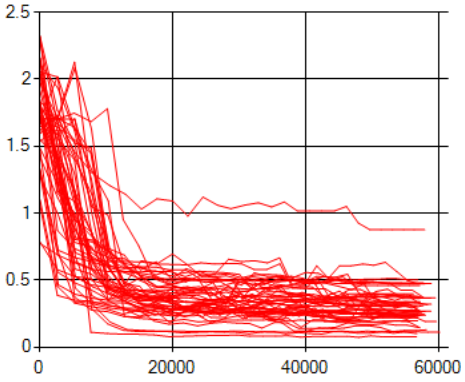


SA_DE_sextic_twister

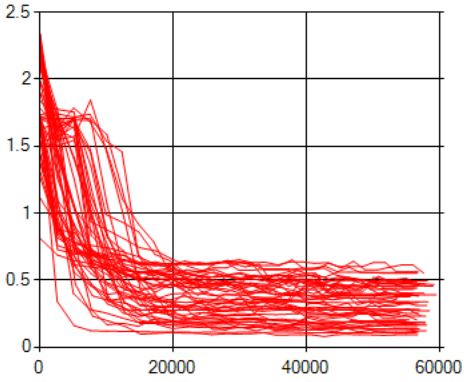


SA_SOMA_sextic_twister

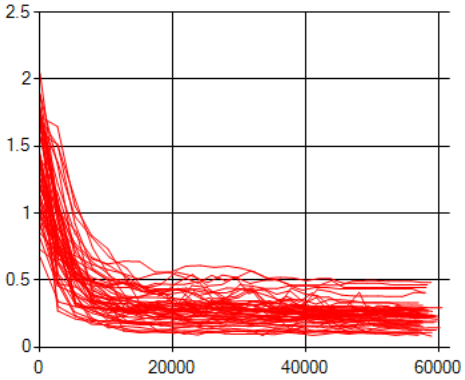
Obrázek D.7: Průběh SA – Sextic problém – MersenneTwister



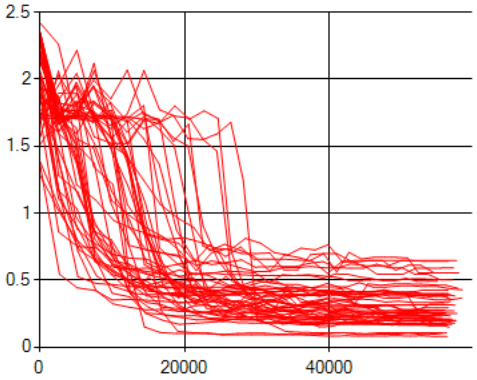
SA_ES_sextic_logistic



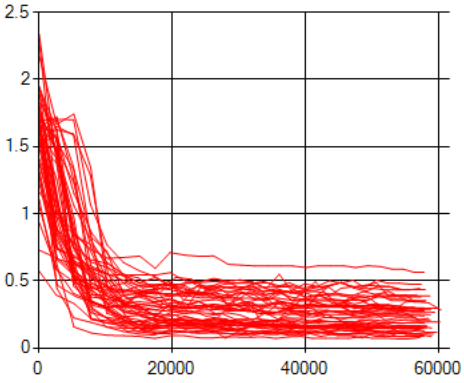
SA_SA_sextic_logistic



SA_PSO_sextic_logistic

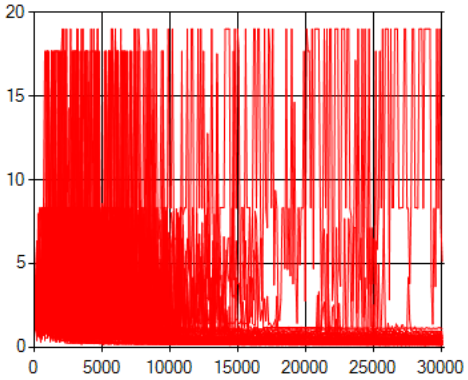


SA_DE_sextic_logistic

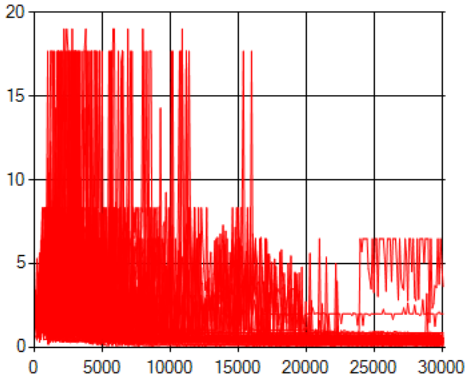


SA_SOMA_sextic_logistic

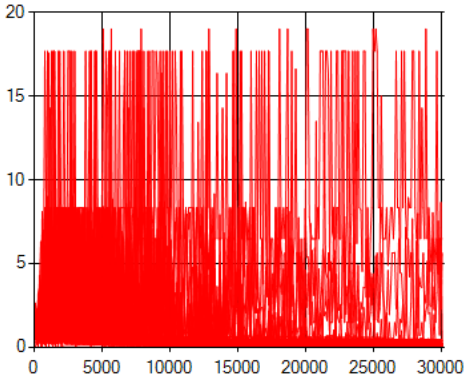
Obrázek D.8: Průběh SA – Sextic problém – LogisticMap



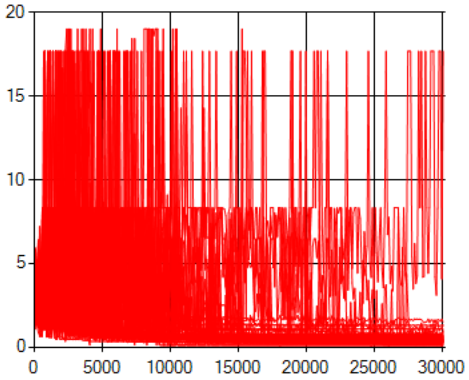
PSO_ES_quintic_twister



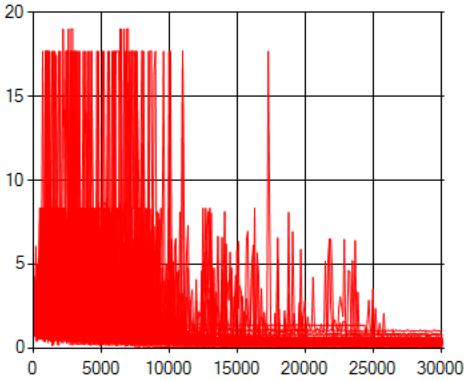
PSO_SA_quintic_twister



PSO_PSO_quintic_twister

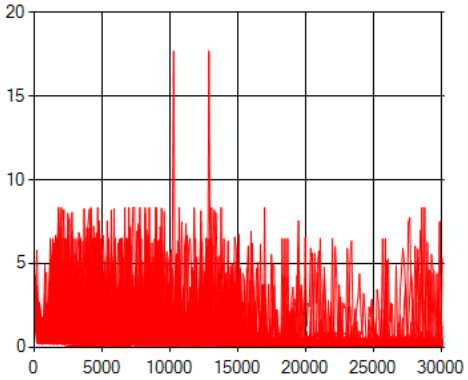


PSO_DE_quintic_twister

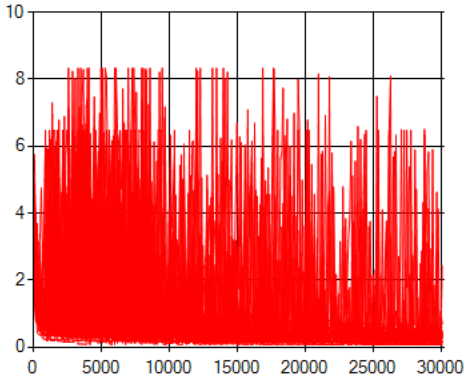


PSO_SOMA_quintic_twister

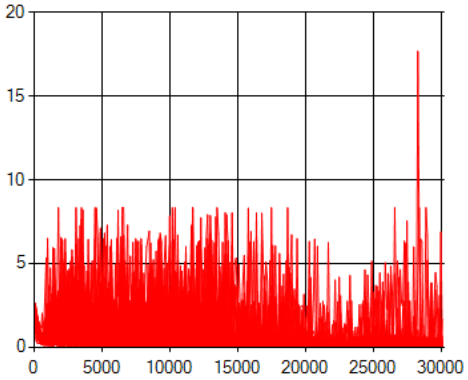
Obrázek D.9: Průběh PSO – Quintic problém – MersenneTwister



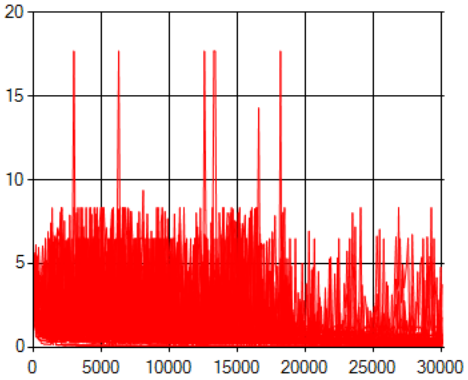
PSO_ES_quintic_logistic



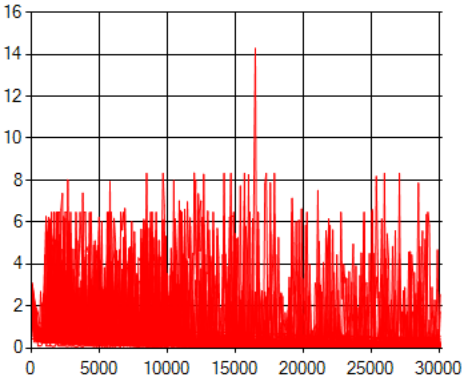
PSO_SA_quintic_logistic



PSO_PSO_quintic_logistic

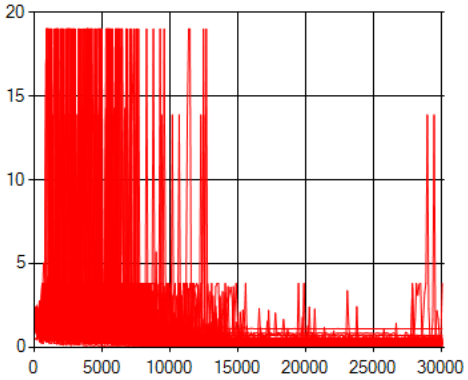


PSO_DE_quintic_logistic

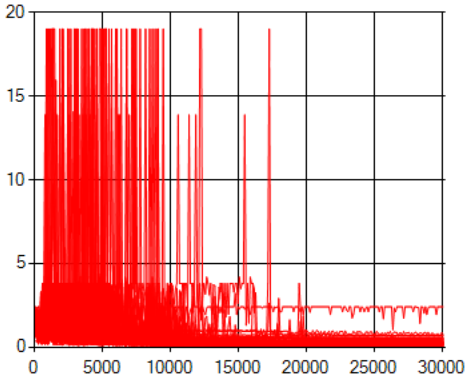


PSO_SOMA_quintic_logistic

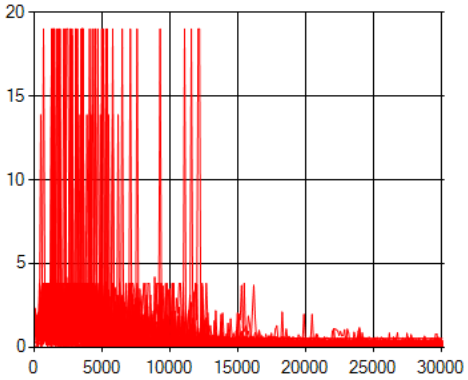
Obrázek D.10: Průběh PSO – Quintic problém – LogisticMap



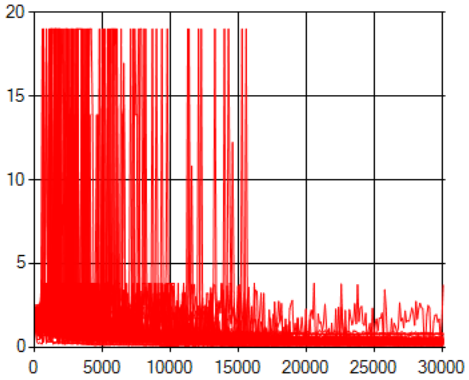
PSO_ES_sextic_twister



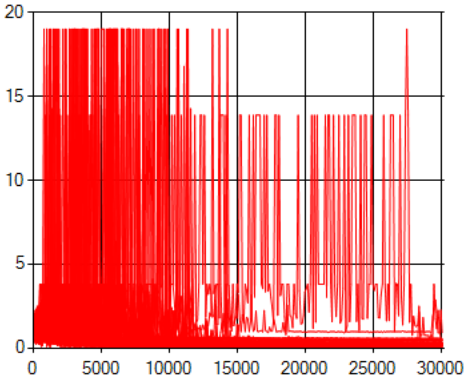
PSO_SA_sextic_twister



PSO_PSO_sextic_twister

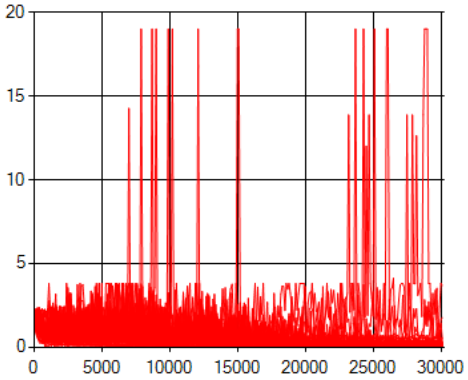


PSO_DE_sextic_twister

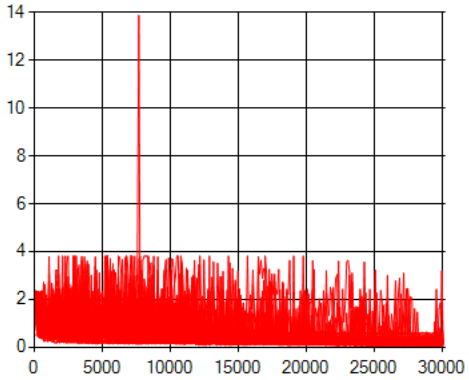


PSO_SOMA_sextic_twister

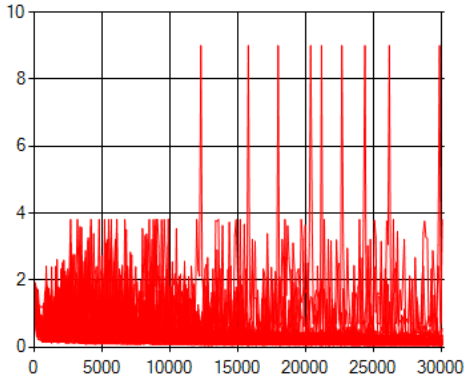
Obrázek D.11: Průběh PSO – Sextic problém – MersenneTwister



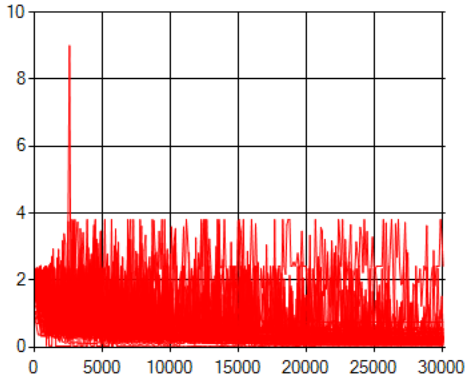
PSO_ES_sextic_logistic



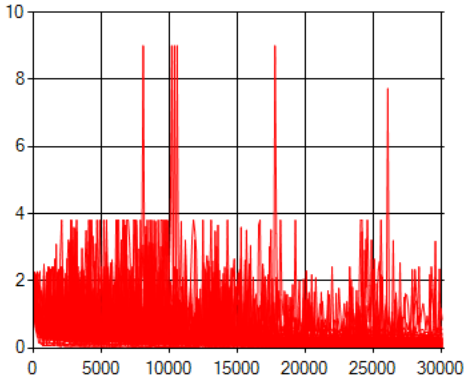
PSO_SA_sextic_logistic



PSO_PSO_sextic_logistic

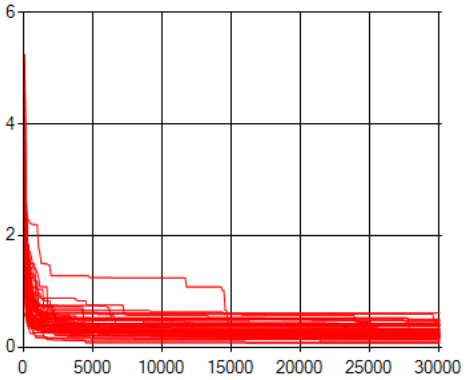


PSO_DE_sextic_logistic

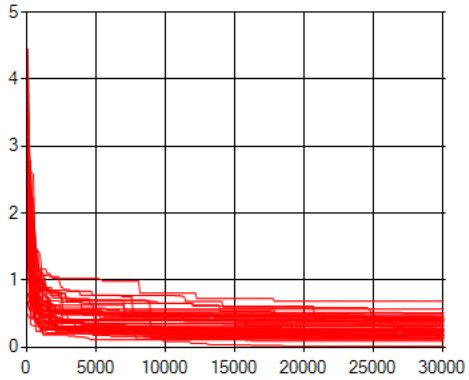


PSO_SOMA_sextic_logistic

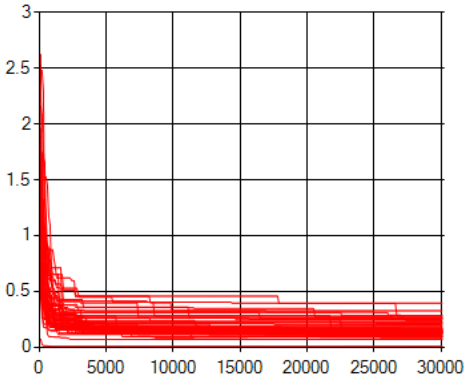
Obrázek D.12: Průběh PSO – Sextic problém – LogisticMap



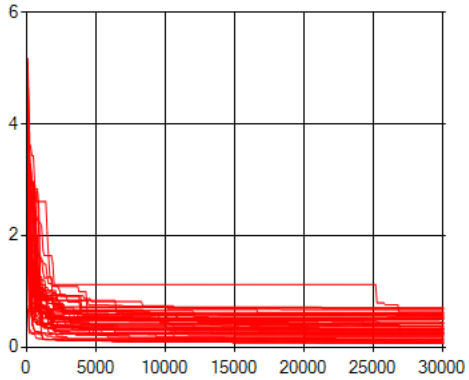
DE_ES_quintic_twister



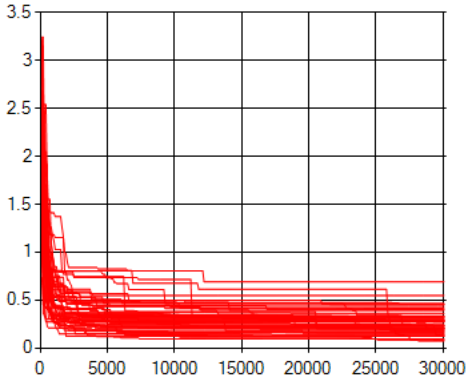
DE_SA_quintic_twister



DE_PSO_quintic_twister

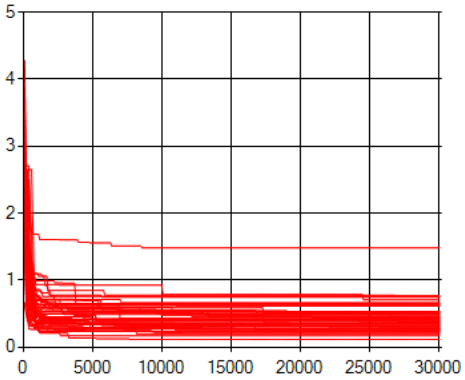


DE_DE_quintic_twister

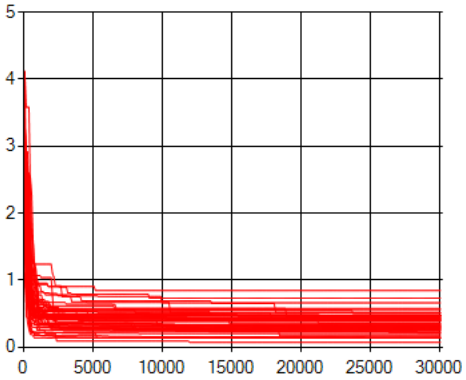


DE_SOMA_quintic_twister

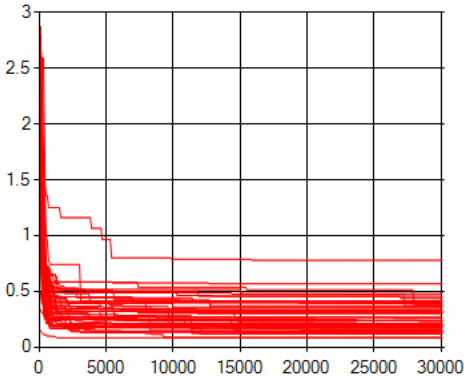
Obrázek D.13: Průběh DE – Quintic problém – MersenneTwister



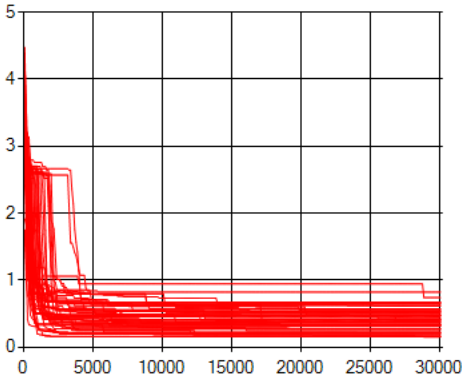
DE_ES_quintic_logistic



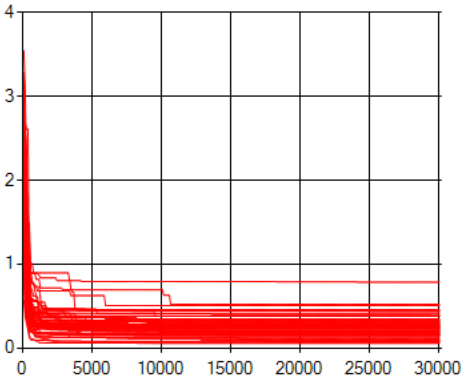
DE_SA_quintic_logistic



DE_PSO_quintic_logistic

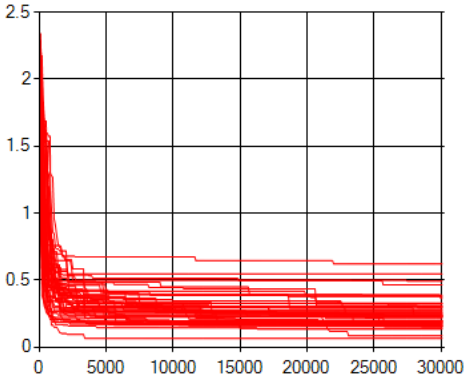


DE_DE_quintic_logistic

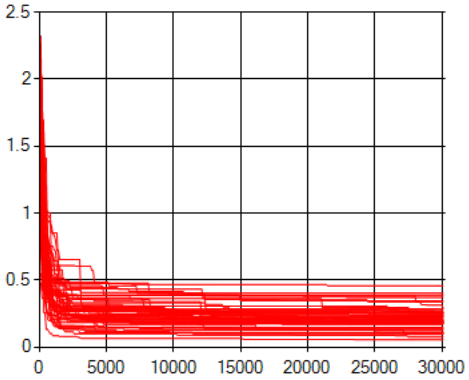


DE_SOMA_quintic_logistic

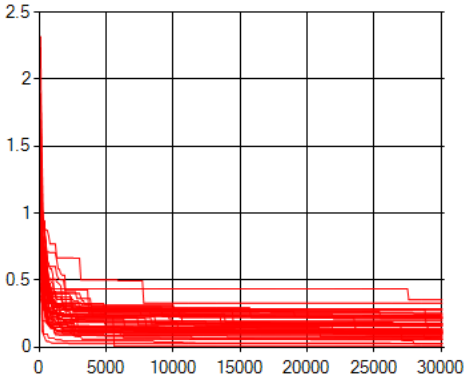
Obrázek D.14: Průběh DE – Quintic problém – LogisticMap



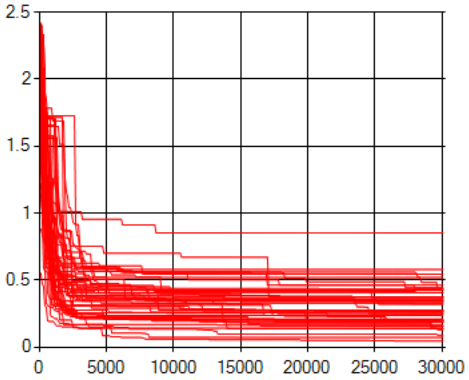
DE_ES_sextic_twister



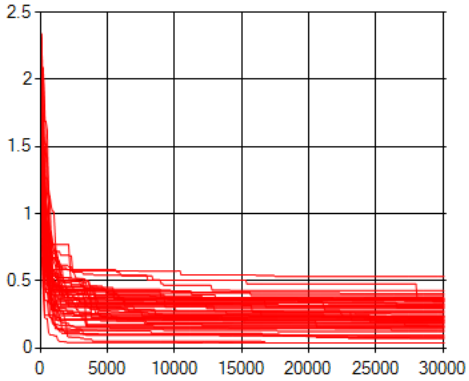
DE_SA_sextic_twister



DE_PSO_sextic_twister

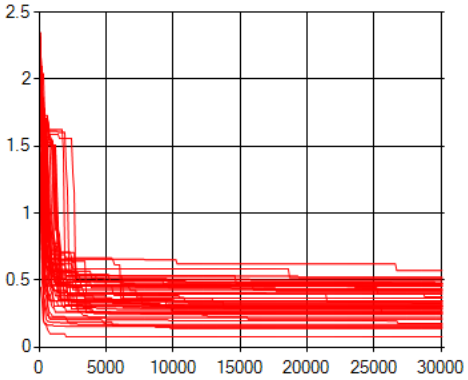


DE_DE_sextic_twister

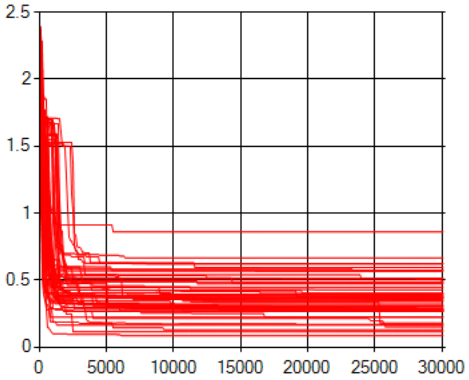


DE_SOMA_sextic_twister

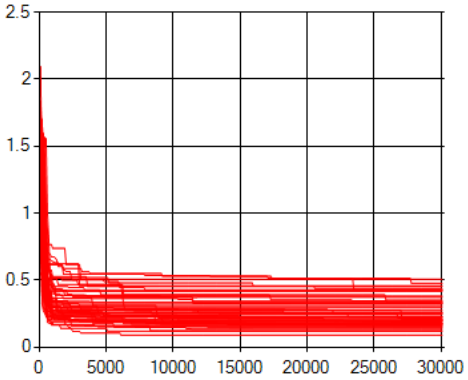
Obrázek D.15: Průběh DE – Sextic problém – MersenneTwister



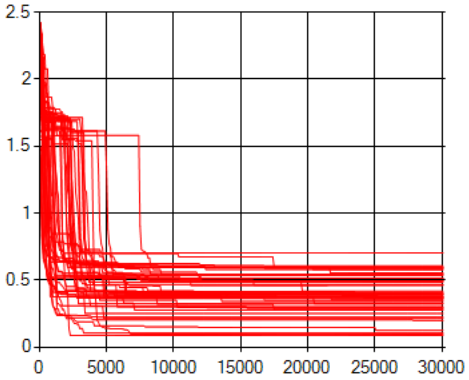
DE_ES_sextic_logistic



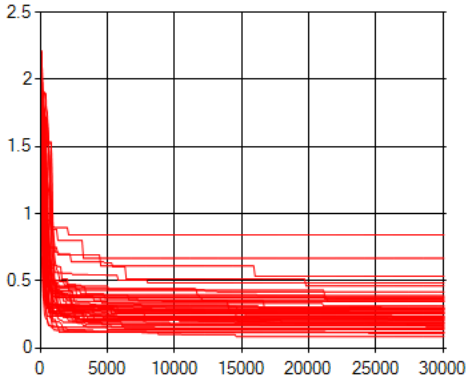
DE_SA_sextic_logistic



DE_PSO_sextic_logistic

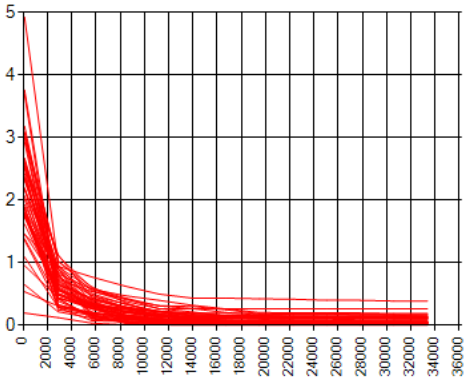


DE_DE_sextic_logistic

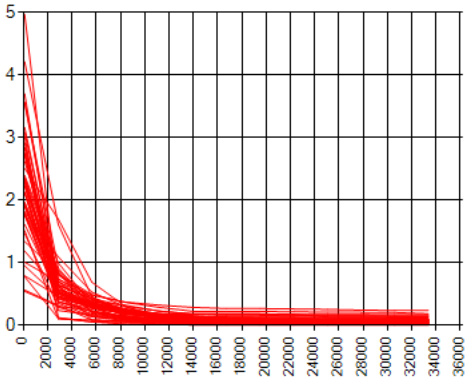


DE_SOMA_sextic_logistic

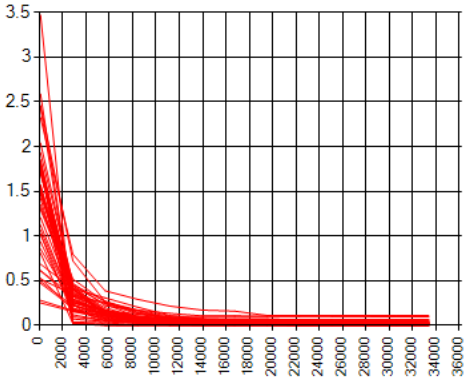
Obrázek D.16: Průběh DE – Sextic problém – LogisticMap



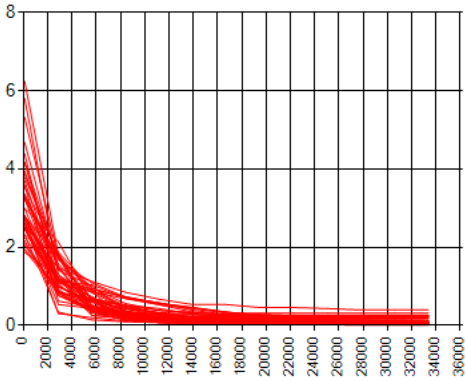
SOMA_ES_quintic_twister



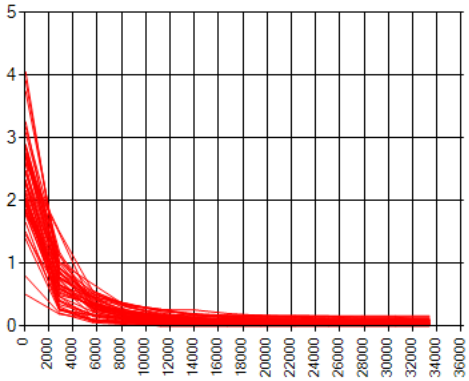
SOMA_SA_quintic_twister



SOMA_PSO_quintic_twister

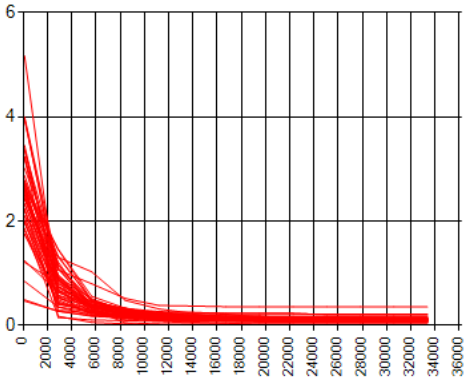


SOMA_DE_quintic_twister

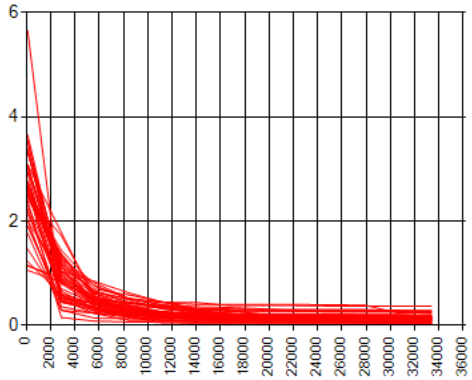


SOMA_SOMA_quintic_twister

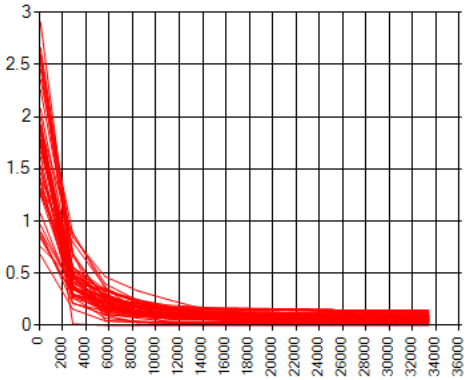
Obrázek D.17: Průběh SOMA – Quintic problém – MersenneTwister



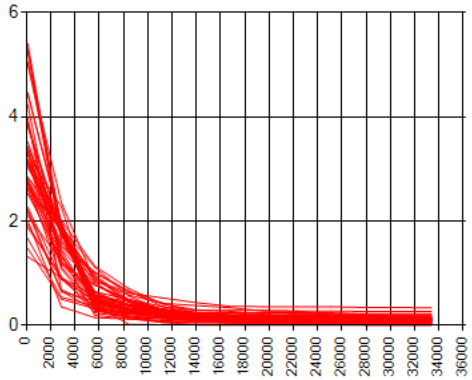
SOMA_ES_quintic_logistic



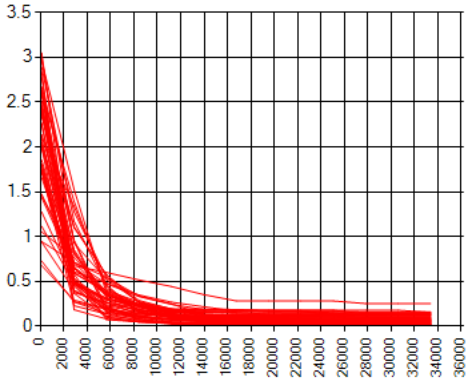
SOMA_SA_quintic_logistic



SOMA_PSO_quintic_logistic

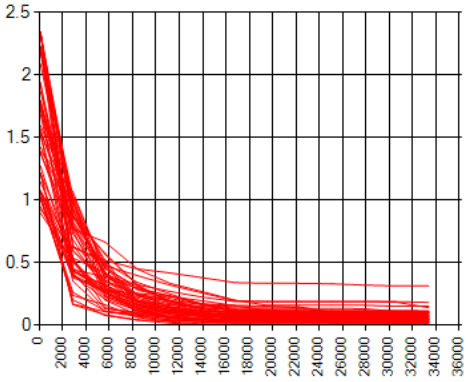


SOMA_DE_quintic_logistic

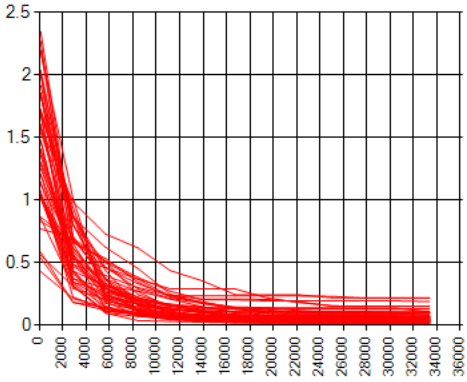


SOMA_SOMA_quintic_logistic

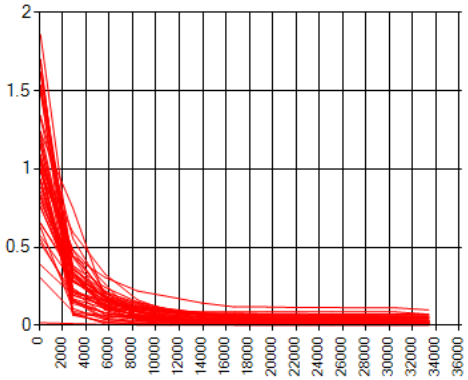
Obrázek D.18: Průběh SOMA – Quintic problém – LogisticMap



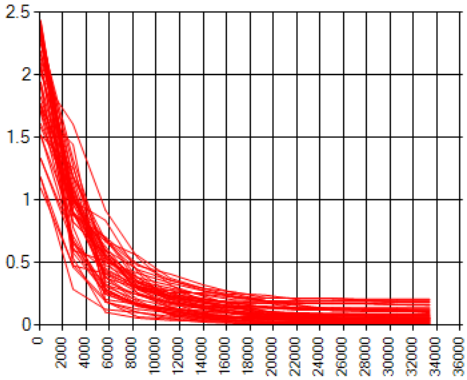
SOMA_ES_sextic_twister



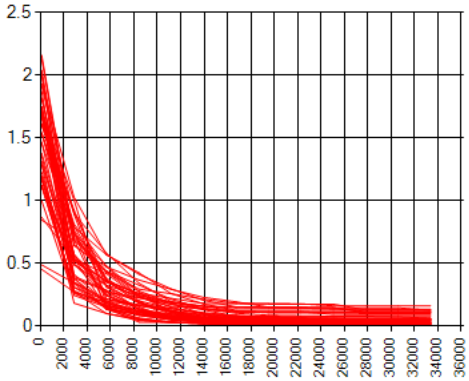
SOMA_SA_sextic_twister



SOMA_PSO_sextic_twister

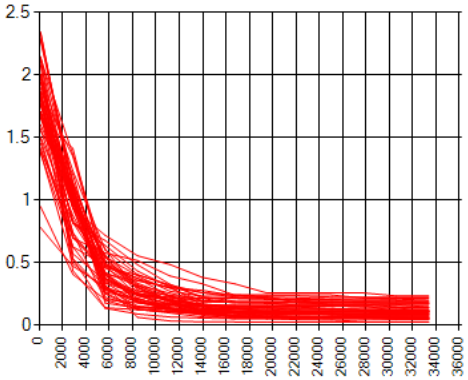


SOMA_DE_sextic_twister

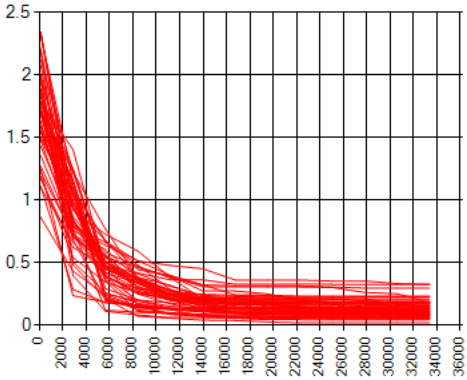


SOMA_SOMA_sextic_twister

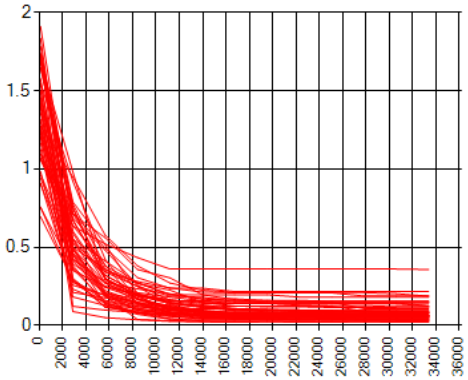
Obrázek D.19: Průběh SOMA – Sextic problém – MersenneTwister



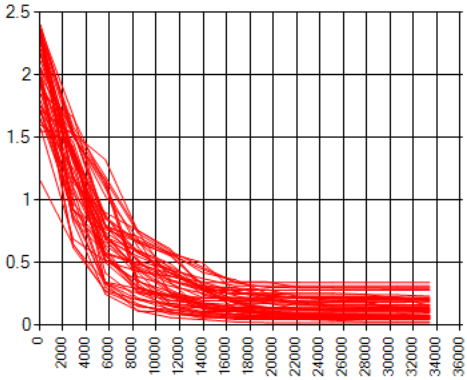
SOMA_ES_sextic_logistic



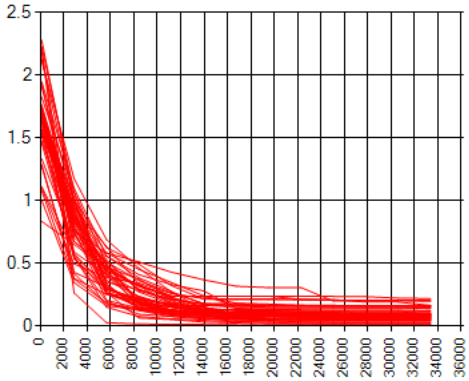
SOMA_SA_sextic_logistic



SOMA_PSO_sextic_logistic



SOMA_DE_sextic_logistic



SOMA_SOMA_sextic_logistic

Obrázek D.20: Průběh SOMA – Sextic problém – LogisticMap